JAO plan wykładu

1 III	Wstęp, wyrażenia regularne, automaty deterministyczne
8 III	Automaty niedeterministyczne, determinizacja, własności domknięcia
15 III	Równoważność automatów i wyrażeń regularnych, lemat o pompowaniu
22 III	Relacja Myhilla-Neroda, minimalizacja
29 III	Problemy decyzyjne, monoidy
5 IV	Automaty dwukierunkowe, gramatyki, drzewa wyprowadzeń
12 IV	Lematy o pompowaniu, postaci normalne, automaty ze stosem
19 IV	brak wykładu
26 IV	Równoważność, deterministyczne automaty
3 V	przerwa
10 V	Algorytm CYK, Tw. Parikha
17 V	Maszyny Turinga, wielotaśmowe, determinizacja
24 V	KOLOKWIUM
31 V	Nierozstrzygalność (m.in. problemu stopu)
7 VI	Nierozstrzygalność ciąg dalszy, P vs NP, NP-zupełność SATa
14 VI	Klasy złożoności

Wykład 1

Technikalia
Moja strona, podstrona przedmiotu. Konsultacje piątek 14:15-15:45, też na maila. Bardzo ważne, żeby zadawać pytania. Zrobię ankietę po kilku wykładach.

Zasady zaliczania. Kolokwium: 40%
Egzamin: 60% - 20% test, 40% zadania
Egzamin poprawkowy: max(poprawkowy 100%, poprawkowy 60% + kolokwium 40%)
Do egzaminów dopuszczeni są wszyscy
Zadania z gwiazdką: 3 serie po około 2-4 zadania, każde zadanie to 100% do podziału po równo między tych co rozwiążą.

Mogę udostępnić nieformalne notatki, które robię dla siebie. Zrobić to?

Tematyka wykładu
Tematyka wykładu: po około ⅓ automaty skończone, gramatyki i automaty ze stosem, maszyny Turinga. Po co ten przedmiot? Kilka odpowiedzi:

- zobaczyć modele obliczeń, do jest głównie przedmiot o modelach obliczeń przez różnorakie maszyny

- automaty skończone, co da się zrobić dysponując skończoną pamięcią, pojawia się to nieraz w informatyce
- maszyny Turinga - formalizują wszystko co może zrobić komputer
- na podstawie tego będzie złożoność obliczeniowa, która klasyfikuje które problemy są łatwe, a które nie dadzą się rozwiązać w rozsądnym czasie - na koniec zrobimy trochę tego
- techniki z teorii automatów przydają się w innych dziedzinach, ja miałem publikacje z dziedziny teorii baz danych i z dziedziny weryfikacji programów, które de facto bazowały na teorii automatów

Słowa

Dla nas słowo to będzie ciąg znaków z ustalonego (zwykle skończonego) alfabetu Sigma. Wszystko z zasadzie można zapisać jako słowo, stąd pracujemy na słowach. Zwykle będzie nas interesować podział zbioru słów na dobre i niedobre. Czyli innymi słowy będziemy chcieli zdefiniować pewien zbiór słów. Zbiór słów nazywamy językiem. Przykłady z życia: poprawny numer PESEL, konta w banku, ale też zbiór ciągów 0-1 takich, że zakodowana liczba jest pierwsza albo zbiór wejść do algorytmu szukającego ścieżki z x do y takich, że algorytm odpowie TAK. Różne są języki, my na początek będziemy się zajmowali tymi prostszymi. Język odpowiada problemowi decyzyjnemu.

Słowo puste (0 liter) będziemy oznaczać przez epsilon. Długość słowa to liczba liter. Konkatenacja słów to napisanie jednego za drugim, oznaczamy kropką u \cdot v, ale często piszemy po prostu uv. Definicja prefiksu, infiksu, sufiksu.

Wyrażenia regularne

Zdefiniujemy **wyrażenia regularne**, prosty sposób wyrażenia prostych języków. Będą one opisywały języki regularne.

Języki regularne mają bardzo fajne własności:

- prosta klasa zawierająca wiele naturalnych języków
- ma wiele alternatywnych definicji (typowa sytuacja w matematyce, która wskazuje na to, że mamy do czynienia z ważnym i właściwym pojęciem)
- zamknięte na wiele naturalnych operacji

Chciałoby się móc powiedzieć:

- nic nie należy do języka
- ustalone słowo należy do języka,
- możemy zapisać sumę dwóch zdefiniowanych języków, i bardziej skomplikowane
- jak zdefiniujemy języki K i L (tak zwykle oznaczamy języki) to umiemy też zdefiniować język K L, który składa się z dwóch połówek: pierwszej w K, drugiej w L
- jak zdefiniujemy język L, to umiemy też zdefiniować język L* (gwiazdka Kleene'go), który składa się z wielu kawałków, z których każdy należy do L
- podobnie z językiem L+

A więc formalnie języki regularne to (najmniejsza) klasa języków, która:

- zawiera zbiór pusty
- zawiera słowo epsilon
- zawiera słowo a dla każdej litery a w Sigma
- jest zamknięta na sumę
- jest zamknięta na konkatenację
- jest zamknięta na gwiazdkę (Kleenego)

Jeden ze sposobów zapisu (później poznamy fundamentalnie inne podejście) języków regularnych jest taki:

- zapisujemy sumę jako +
- konkatenację jako kropkę
- gwiazdkę jako *

Przykłady

a* - zbiór słów złożonych z samych liter a
(a+b)* - zbiór słów złożonych z liter a i b
abac - singleton słowa abac
(ab)* - zbiór słów, które są postaci ab ileś razy
(ab)*c - to co wyżej i jeszcze zakończone na c
(ab)* + b(ab)* + (ba)* + a(ba)* - litery a i b na zmianę, albo (a+eps)(ba)*(b+eps)

A jak zrobić: zaczynają się na a, kończą na b?
a(a+b)*b, a może a(a+b+c)*b? To zależy od alfabetu, zawsze precyzujemy nad jakim alfabetem pracujemy. I piszemy wtedy aSigma*b.

A jak powiedzieć: zbiór słów o ustalonym prefiksie / sufiksie / infiksie?
wSigma*, Sigma*w, Sigma* w Sigma*

A pierwsza litera równa ostatniej? aSigma*a + bSigma*b + cSigma*c

A jak zrobić: nie ma infiksu aa? (a + eps)(ba)*(b + eps)
A jak zrobić: nie ma infiksu aab? To już będzie trudniej, chociaż da się.

Widać, że w ten sposób możemy zrobić wszystkie języki regularne po prostu konstruując wyrażenie.

Automaty skończone

Teraz przejdziemy do dużo bardziej naturalnego pojęcia, **automatów skończonych**.
Automaty skończone są innym sposobem opisu języków regularnych, dużo bardziej wygodnym.
Przykład: słowa o parzystej liczbie a i parzystej liczbie b.
Wyrażenie regularne: (aa + bb + (ab + ba)(aa + bb)*(ab + ba))*
Automat: łatwy do zrobienia. Pokazuję na przykładzie jak to wygląda.

Formalnie: deterministyczny automat skończony (deterministic finite automaton - DFA) A = (Q, Sigma, q0, F, delta) składa się ze skończonego zbioru stanów Q,
jest nad skończonym alfabetem Sigma, z wyróżnionym stanem początkowym q0, zbiorem stanów akceptujących F oraz funkcją przejścia delta: Q x Sigma -> Q.
Bieg po w = a1 … an to ciąg tranzycji qi \trans{a_i} q_{i+1} takich, że q1 jest początkowy.
Bieg jest akceptujący jeśli q_{n+1} in F. Automat akceptuje słowo w jeśli bieg po w jest akceptujący. Zbiór słów, które automat akceptuje to jego język, ozn. L(A), mówimy, że A rozpoznaje L(A).
Notacja: nieraz piszemy w takiej sytuacji q1 \trans{w} q_{n+1}.

Idea jest taka, że automaty to po prostu coś ze skończoną pamięcią. Okazuje się, że języki, które są rozpoznawane przez DFA to dokładnie języki regularne.

Tw. Kleene'ego (1956)
Język jest rozpoznawalny przez DFA wtw. gdy jest regularny.

Dowód będzie na wykładzie 3.

Teraz zobaczmy kilka przykładów automatów.
Automaty dla:
- (ab)*
- (aaa)*
- pierwsza litera równa ostatniej (zrobiliśmy dotąd)
- parzystość a = parzystość b
- jest podciąg aab
- nie ma podciągu aab

Wykład 2

Automaty niedeterministyczne
Teraz spojrzymy na nieco mocniejszy model, niedeterministyczny automat skończony (NFA). Różnica jest taka, że on może mieć wiele stanów początkowych (może zacząć w jednym z wielu miejsc) i może mieć wiele tranzycji z tego samego stanu po tej samej literze.

Przykład: Sigma*abc

Formalnie: A = (Q, Sigma, I, F, delta), gdzie
Q - skończony zbiór stanów
Sigma - skończony alfabet
I \subseteq Q - stany początkowe
F - subseteq Q - stany akceptujące
delta - **relacja** przejścia, delta \subseteq Q x Sigma x Q

Bieg - jak poprzednio ciąg tranzycji, zaczyna się w stanie początkowym
Bieg jest akceptując jeśli kończy się w stanie końcowym.
Teraz: po jednym słowie może być wiele biegów.
Słowo jest akceptowane przez automat jeśli istnieje (chociaż jeden) bieg akceptujący.
Język A to zbiór słów akceptowanych.

Powiemy, że automaty są równoważne jeśli mają te same języki.
Pytanie: czy dla każdego automatu niedeterministycznego istnieje równoważny automat deterministyczny? Ankieta studentów. Odpowiedź: TAK.

Determinizacja
Tw.
Dla każdego automatu niedeterministycznego istnieje równoważny automat deterministyczny.

Dowód
Weźmy NFA A = (Q, Sigma, I, F, delta)
Zrobimy równoważny DFA A' = (Q', Sigma, q0', F', delta')
Idea ogólna DFA jest taka, że po prostu pamiętamy wszystko to, co jest nam potrzebne o słowie, które wczytaliśmy do tej pory i dla każdego stanu pamięci mamy jeden stan.
Idea: patrzymy gdzie może A pójść po słowie w, do których stanów i to pamiętamy. Czyli innymi słowy pamiętamy podzbiór stanów.
Q' = P(Q)
q0' = {I}
F' = {S | S \cap F \neq \emptyset}
delta'(S, a) = {q | exists_{p in S} (p, a, q) in delta}

Dlaczego to jest ok. Pokażmy, że istotnie L(A) = L(A'). Inkluzja L(A) \subseteq L(A') - jeśli jest bieg po w, to po tym w idziemy z I do jakiegoś S i w tym S jest stan akceptujący. Inkluzja L(A') \subseteq L(A) - jeśli jest bieg po w, to znaczy, że z pewnego stanu z I do się dojść po w do pewnego stanu z F, koniec dowodu. Komentarz: w zasadzie to jest masło maślane, widać, że A' dobrze symuluje A.
c.n.d.

Przykład dla Sigma* abc
Widać, że wszystkie stany osiągalne A' będą zawierały stan początkowy A, więc wystarczy rozważyć ich 8.

Uwaga: determinizacja może być wykładnicza. Przykład: n-ta litera od końca to a.
NFA może to zrobić używając n+1 stanów. **Pytanie**: ile potrzebuje DFA?
Co najmniej 2^n. Formalnie będziemy umieli to zrobić za 1-2 wykłady, ale teraz zobaczmy z grubsza dlaczego. Idea: musi pamiętać ostatnie n liter, bo gdyby coś zapomniał, to mogłoby się okazać, że to właśnie jest ważne. Czyli potrzeba 2^n stanów, no i tyle wystarczy.

Operacje na językach

Będziemy wymiennie używali automatów deterministycznych i niedeterministycznych, bo wiemy, że ich wyrażalność jest taka sama. Natomiast nie używamy na razie Tw. Kleene'go o tym, że automaty (DFA lub NFA) mają tą samą wyrażalność co wyrażenia regularne, bo tego żeśmy jeszcze nie udowodnili.

Dopełnienie - jak?

Dla automatu deterministycznego dużo łatwiej. Po prostu zamieniamy stany akceptujące i nieakceptujące. Czy to działa? Tak po prostu to nie, musimy mieć własność taką, że po każdym słowie jest bieg. Więc wprowadzamy dodatkowy stan - "śmietnik", do którego idą wszystkie tranzycje, które wcześniej nie istniały. Po tej modyfikacji funkcja przejścia automatu jest już prawdziwą funkcją, a nie tylko funkcją częściową. W konstrukcji dopełnienia ten stan "śmietnik" stanie się akceptujący. W przyszłości będziemy używać automatów ze śmietnikiem lub bez, w zależności od wygody. Zwykle wygodnie jest go nie rysować, ale czasem, jak przy dopełnieniu, warto mieć własność taką, że delta jest funkcją, wtedy go używamy.

Tu ciekawym pytaniem jest: jak duży może być NFA dla \bar{L} (dopełnienia L) w stosunku do NFA dla L. Bo dla DFA są takie same. Pytanie: jak automat ma n stanów to jak długie może być najkrótsze słowo, które akceptuje (o ile akceptuje jakieś)? Oczywiście nie dłuższe niż n-1. Okazuje się, że da się zrobić taki NFA A, że najkrótsze słowo, którego nie akceptuje jest wykładniczej długości względem n. Wtedy w dopełnieniu L(A) najkrótsze akceptowane słowo jest wykładniczej długości względem n, czyli automat musi mieć przynajmniej taką wielkość.

Suma, przecięcie

Suma - robimy automat A = (Q, Sigma, q, F, delta) z automatów A1 = (Q1, Sigma, I1, F1, delta1) oraz A2 = (Q2, Sigma, I2, F2, delta2). Po prostu A będzie rozłączną sumą A1 i A2, czyli formalnie Q = Q1 \cup Q2, I = I1 \cup I2, F = F1 \cup F2, delta = delta1 \cup delta2.

Przecięcie - nie da się już tak łatwo. Idea jest taka, że będziemy symulowali oba automaty i akceptowali jeśli oba zaakceptowały.

Jak wyżej robimy A z A1 i A2. Ustalamy:

- Q = Q1 x Q2 (nowy stan musi pamiętać oba stany)
- I = I1 x I2
- F = F1 x F2
- delta też zachowuje się produktowo, tzn. delta = {(p1,p2)--a→ (q1,q2) | p1 --a→ q1 oraz p2 --a→ q2}

Łatwo zauważyć, że w in L(A) wtw. w in L(A1) oraz w in L(A2), można pokazać implikację w dwie strony.

Zauważmy, że podobnie można też zrobić konstrukcję dla sumy, tyle, że wtedy dajemy F = (F1 x Q2) \cup (Q1 x F2), czyli akceptujemy jeśli co najmniej jeden ze stanów jest akceptujący.

Automaty z epsilon-przejściami
Wprowadzimy teraz nowy model automatu (ale nie bardzo różny), który będzie przydatny do różnych technicznych rozważań. Okazuje się, że będzie miał tę samą wyrażalność co DFA i NFA. Najpierw przykład: (ab)* (aaa)* (dwa automaty połączone epsilon-przejściem).
Automat z epsilon-przejściami to po prostu NFA, który może mieć puste słowo na tranzycjach. Jego język jest zdefiniowany normalnie, tyle, że słowo na ścieżce po prostu czyta epsilony.
Oczywiście automaty z epsilon-przejściami akceptują wszystkie języki akceptowane przez NFA, bo po prostu każdy NFA jest też automatem z epsilon-przejściami (tyle, że zeroma).
Pytanie jak w drugą stronę, czy mogą coś więcej? **Pytanie** do sali. Odpowiedź: NIE.

Tw.
Automaty z epsilon-przejściami akceptują te same języki co automaty niedeterministyczne (i deterministyczne też zresztą).

Dowód
Przerabiamy automat z eps-przejściami na automat równoważny (o tym samym języku), ale taki, który nie ma już eps-przejść. Tzn. eliminujemy eps-przejścia. Najpierw dla każdej ścieżki typu p1 --eps→ p2 --eps→ … --eps→ pk --a→ q1 --eps→ … --eps→ qm dodajemy nową tranzycję p1 --a→ qm. Nowe przejścia nie zmieniają języka. Dla każdego akceptowanego niepustego słowa znajdziemy bieg, który nie używa eps-przejść. Więc prawie możemy teraz usunąć eps-przejścia, trzeba tylko uważać na puste słowo. No to usuwamy eps-przejścia i patrzymy czy zmieniła się sytuacja pustego słowa. Jeśli tak, to dodajemy je sztucznie (dodając nowy stan równocześnie początkowy i akceptujący).
c.n.d.

Wykład 3

Dowód tw. Kleene'go
Jesteśmy już gotowi, żeby udowodnić równoważność wyrażeń regularnych i automatów. Przypomnijmy, że to właśnie mówi twierdzenie Kleene'go (1956).
Potrzeba pokazać, że dla każdego wyrażenia regularnego r istnieje automat A taki, że L(A) = L(r) oraz odwrotnie, że dla każdego automatu A istnieje wyrażenie regularne r takie, że L(r) = L(A).

Zaczniemy od przerabiania wyrażenia na automat, czyli pokazywania, że dla każdego wyrażenia r istnieje automat A taki, że L(A) = L(r). Będziemy to pokazywać indukcyjnie po budowie wyrażenia regularnego. Czyli zaczniemy od wyrażeń bazowych, tj.: puste, epsilon, a, a potem będziemy pokazywać dla coraz większych wyrażeń. Pokażemy innymi słowy automaty dla wyrażeń: puste, epsilon, a, a potem pokażemy, że jeśli mamy dla wyrażeń r1 i r2 mamy odpowiednio automaty A1 i A2, to umiemy zbudować automat dla wyrażeń r1+r2, r1 r2, r1*.
W ten sposób możemy składać z małych wyrażeń coraz większe, czyli formalnie właśnie dowodzić indukcyjnie po budowie wyrażenia.

Automaty dla wyrażeń: puste, epsilon, a są banalne (narysować).

Zajmijmy się sumą. Będziemy robić automat z epsilon-tranzycjami dla L(A1) \cup L(A2). **Pytanie** do wszystkich - jak to zrobić? Łatwo - robię nowy stan początkowy i z niego eps-tranzycje do wszystkich stanów początkowym w A1 i w A2 (czyli odpowiednio z I1 i z I2).

A jak dla konkatenacji, czyli dla L(A1) L(A2)? Też łatwo - dodaję eps-tranzycje pomiędzy każdym stanem akceptującym w A1 (czyli z F1) a każdym stanem początkowym w A2 (czyli z I2). Pokazać, że to istotnie to samo.

A dla L(A1)*? Dodaję eps-tranzycje z każdego stanu akceptującego (z F1) do każdego stanu początkowego (z I1). Czy to jest ok - **pytanie** do sali. Nie, ale prawie. W L(A1)* powinno być słowo puste, ta konstrukcje niekoniecznie o to dba. Dodajemy więc niezależnie słowo puste (nowy stan akceptujący i eps-przejście do niego).

To kończy dowód w jedną (nieco prostszą) stronę.
Zasymulujmy jak to działa na przykładzie wyrażenia (a+b+c)*c(a+b)* + (a+c)*b(a+b+c)*.

Teraz robimy w drugą stronę, mianowicie dla danego automatu budujemy równoważne wyrażenie. Na początek zakładamy, że nasz automat ma dokładnie jeden stan akceptujący, język każdego automatu deterministycznego jest skończoną sumą takich właśnie automatów, po prostu w każdym z automatów zostawiamy jako akceptujący tylko jeden ze stanów akceptujących w oryginalnym automacie. Nasz plan postępowania jest następujący: bierzemy automat i będziemy po kolei zmniejszać liczbę jego stanów pisząc na krawędziach nie tylko litery, ale również wyrażenia. Na koniec dostaniemy automat z dwoma stanami i jakimś wyrażeniem na krawędzi, to będzie interesujące nas wyrażenie albo z jednym stanem (i wtedy szukane wyrażenia to gwiazdka z wyrażenia na pętli). Formalnie rzecz biorąc wprowadzamy tutaj znów nowy model automatu, tj. automat, który na krawędziach ma napisane nie tylko litery albo słowa puste, ale wyrażenia. Wtedy przejście jest po dowolnym słowie należącym do języka wyrażenia. Nie będziemy robić tego formalnie, bo właśnie powiedzieliśmy precyzyjnie o co chodzi, a nie chodzi o formalność, tylko o precyzję. **Pytanie**: czy ktoś chce, żeby to napisać nieco dokładniej?

No to teraz jak przerabiamy? Zrobić to na przykładzie automatu dla: parzysta liczba a i parzysta liczba b. Wychodzi nieco brzydko, ale da się zasymulować. W ogólności: jeśli eliminujemy stan q, to dla dowolnych stanów p1, p2 jeśli wcześniej było:

- p1 --r1→ q
- q --r2 → p2
- q --r3→ q
- p1 --r4→ p2

To teraz piszemy p1 --(r4 + r1(r3)*r2)--> p2. No bo tak właśnie mogliśmy dojść z p1 do p2 (nie przechodząc przez q, bądź przechodząc przez nie). Pokazujemy, że jeden krok tej modyfikacji nie zmienia języka, co kończy dowód.
c.n.d.

Lemat o pompowaniu
Teraz już wiemy, że wszystkie automaty i wyrażenia, które badaliśmy opisują tę samą klasę języków: języki regularne. Fajnie byłoby coś zrozumieć o tych językach. W szczególności dla ustalonego języka dobrze byłoby umieć pokazać, że on nie jest regularny. Mówiliśmy już, że intuicyjnie L = {w | #a(w) = #b(w)} nie jest regularny (bo trzeba by pamiętać różnicę pomiędzy liczbą a, a liczbą b do tej pory, a to może być dowolnie duża), ale chcemy to umieć sformalizować. Teraz stworzymy pierwsze narzędzie do tego: lemat o pompowaniu. Niedługo nauczymy się drugiego (moim zdaniem wygodniejszego) narzędzia, ale zacznijmy od tego.

Lemat o pompowaniu ma to do siebie, że łatwiej zapamiętać jego dowód niż jego sformułowania. Zresztą (moim zdaniem) dowód jest też bardziej interesujący niż sformułowanie. Dlatego polecam zapamiętania dowodu, bo wtedy łatwiej odtworzyć treść lematu.

Załóżmy, że mamy automat deterministyczny o n stanach.
Najpierw obserwacja: jeśli mamy bieg po słowie w długości co najmniej n, to odwiedza on przynajmniej n+1 stanów, więc z zasady szufladkowej Dirichleta któryś stan zostanie odwiedzony przynajmniej 2 razy. A więc możemy przedstawić nasz bieg w postaci:

p --s→ q --t→ q --u→ r,

gdzie w = stu. No ale pętlę q --t→ q możemy powtarzać wiele razy, więc dowolne słowo postaci st^k u również będzie miało bieg akceptujący, czyli będzie należało do języka. Co więcej możemy wybrać słowa s i t tak, żeby |st| \leq n, bo już na prefiksie słowa w o długości n powtórzy się jakiś stan q.
To już właściwie koniec dowodu, ale najpierw sformułujmy lemat.

Lemat o pompowaniu
Dla każdego język regularnego L istnieje n in N takie, że dla każdego słowa w o długości co najmniej n istnieją słowa s, t, u takie, że:
- w = stu
- |st| \leq n, t \neq epsilon
- st^k u in L dla dowolnego k in N

Dowód
Bierzemy n równe liczbie stanów automatu deterministycznego (choć to niekonieczne) dla L. Wtedy lemat wynika z obserwacji powyżej.
c.n.d.

Przykłady zastosowania lematu
Przykład 1: rozważmy język L = {w | #a(w) = #b(w)}. Przypuśćmy, że jest regularny. Wówczas istnieje dla niego pewna liczba n z lematu o pompowanu.
Weźmy w = a^n b^n \in L. Wtedy w = stu. Ponieważ |st| \leq n, to musi być t = a^i, i > 0. Zatem st^2u = a^{n+i} b^n też należy do L (z lematu). No ale ewidentnie a^{n+i} b^i nie należy do L, sprzeczność.

Przykład 2: niech L = {a^i b^j | i \geq j}. Przypuśćmy, że L jest regularny. Wówczas istnieje dla niego n z lematu o pompowaniu. Weźmy w = a^n b^n \in L. Wtedy w = stu, podobnie jak wcześniej mamy t = a^i, i > 0. Ale jak zbadamy st^2u, to on naprawdę należy do L. Możemy jednak wziąć k = 0. I wtedy mamy, że su = a^{n-i} b^n też należy do L, ale jednak nie należy. Sprzeczność.

Przykład 3: niech L = język palindromów. Bierzemy w = a^n b a^n, pompujemy, działa.

Przykład 4: niech L = {w | #a(w) = #b(w) = 2#c(w)}. Można ręcznie. Ale można też: gdyby był automat dla L, to automat, w którym wszystkie c zamieniamy na epsilon rozpoznawałby {w | #a(w) = #b(w)}, a ten już wiemy, że jest nieregularny.

Przykład 5: niech L = {a^i b^j c^k d^m | i + j = k + l}. Języki regularne są zamknięte na przecięcie, więc przetnijmy L z a*c*. Jeśli tamten był regularny, to nowy też będzie. Ale L \cap L(a*c*) = {a^i c^i}, który jest nieregularny jak łatwo pokazujemy używając lematu o pompowaniu.

Uwaga: to nie jest tak, że język jest regularny wtedy i tylko wtedy, gdy spełnia lemat o pompowaniu. Są języki nieregularne, które spełniają lemat o pompowaniu.
Przykładem może być język: L = {c a^n b^n} \cup {c^k w | w zaczyna się nie na c, k \neq 1}. Druga część jest regularna, więc może być napompowana, a pierwsza część może być napompowana przez napompowanie lub odpompowanie c i wpadnięcie do drugiej części. Czyli L spełnia lemat o pompowaniu chociaż nie jest regularny (co się okaże później, ale intuicyjnie już widać).

Uwaga: można sobie dowodzić własne wersje lematu. Dużo bardziej należy z tego zapamiętać technikę usuwania/dodawania kawałków pomiędzy równymi konfiguracjami niż sformułowania lematu!

Wykład 4

Relacja Myhilla-Neroda (John Myhill, Anil Nerode)
Teraz zajmiemy się pojęciem, które da nam inne narzędzie do sprawdzania, czy język jest regularny. Poza tym okaże się przydatne np. do zmniejszania rozmiaru automatu deterministycznego przy zachowaniu jego języka. Pomysł bierze się za następującego faktu. Ustalmy automat deterministyczny A, niech L = L(A). Powiedzmy, że dwa słowa: u i v prowadzą do tego samego stanu q w automacie A. Wówczas mają one tę samą przyszłość, to znaczy uw

\in L ⇔ vw \in L, bo to po prostu znaczy, że słowo w przechodzi ze stanu q do jakiegoś stanu akceptującego. To motywuje następującą definicję. Dla ustalonego języka L (niekoniecznie regularnego) relacja równoważności \sim_L na zbiorze słów Sigma^* jest zdefiniowana jako:

u \sim_L v wtw. dla każdego słowa w zachodzi uw \in L ⇔ vw \in L

Mówimy o niej relacja Myhilla-Neroda. Intuicyjnie znaczy to, że u i v mają te same przyszłości. Uwaga: inaczej można napisać, że u^{-1} L = v^{-1} L, to oznacza to samo. Formalnie u^{-1} L nazywa się ilorazem lewostronnym języka. Łatwo zauważyć, że \sim_L jest relacją równoważności. Jej istota ujęta jest w następującym twierdzeniu.

Zakładamy, że funkcja przejścia automatu musi być funkcją całkowitą. Ta definicja jest jednak lepsza. Nieraz rysujemy automaty bez stanu śmietnik, ale formalnie powinien on tam być.

Tw. Myhilla-Neroda (1958)
L jest regularny wtw. relacja \sim_L ma skończenie wiele klas abstrakcji (tzn. skończony indeks)

Dowód:
Najpierw pokażemy implikację w prawo. Jest łatwa. Niech A to DFA taki, że L = L(A).
Jak powiedzieliśmy wcześniej jeśli biegi po u i v prowadzą do tego samego stanu w A, to u \sim_L v. Czyli klas abstrakcji jest co najwyżej tyle, co stanów A, czyli skończenie wiele.

Teraz pokażemy implikację w lewo. Załóżmy, że \sim_L ma skończenie wiele klas abstrakcji. Zauważmy, że jeśli u \sim_L v, to także ua \sim_L va dla dowolnej litery a. Zdefiniujmy następujący automat.

Stanami są klasy abstrakcji \sim_L. Przejścia są zdefiniowane jako [u] --a→ [ua]. Jak pokazaliśmy powyżej to nie zależy od wyboru reprezentanta klasy (czyli u), czyli automat jest deterministyczny. Stan początkowy to [eps], a stany końcowe to [u] dla u \in L. Zauważmy, że w klasach akceptujących wszystkie słowa są akceptujące.
Teraz zobaczmy, że ten automat rozpoznaje język L. Dla dowolnego słowa u dojdziemy w nim do [u]. Jeśli u in L, to ten stan jest akceptujący, a wpp. nie, czyli rzeczywiście dokładnie słowa z L zostaną zaakceptowane.
c.n.d.

Pytanie: czy to samo by zachodziło dla ilorazów prawostronnych języka L? Tak, tu wszystko jest symetryczne. L regularny ⇔ reverse(L) regularny.

Przykłady
Łatwo teraz pokazywać używając tego narzędzia, że dany język nie jest regularny.
Przykład 1: {a^n b^n}. Słowa a^i są nie w relacji M-N. Zobaczmy, że a^i oraz a^j nie są w relacji. Dla sufiksu a^i mamy a^i b^i \in L, ale a^j b^i \not \in L.

Przykład 2: {w | #a(w) = #b(w)}. Ten sam przykład.

Przykład 3: weźmy przykład języka nieregularnego, dla którego działał lemat o pompowaniu, czyli L = {ca^n b^n} \cup {c^k w | w zaczyna się nie na c, k \neq 1}. Pokażemy, że słowa ca^i dla i > 0 są nie w relacji. Faktycznie, ca^i oraz ca^j, dla i \neq j, rozróżniają się sufiksem b^i.

Automat minimalny
Relacja M-N jest też mocno związana z minimalnym automatem deterministycznym. Zauważmy, że pokazaliśmy, że jeśli A to DFA oraz L = L(A), to liczba stanów \geq indeks(\sim_L). Z drugiej strony pokazaliśmy, że da się zbudować DFA A' taki, że L(A') = L(A) oraz liczba stanów A = indeks(\sim_L)'. Czyli innymi słowy pokazaliśmy, że najmniejszy pod względem liczby stanów automat deterministyczny dla L ma ich dokładnie indeks(\sim_L). To odpowiada intuicji: to jest dokładnie ta informacja, którą musimy pamiętać.

Fakt
Minimalny automat deterministyczny dla L ma indeks(\sim_L) stanów.

Zauważmy, że łatwo w ten sposób sprawdzić, że automat jest minimalny. Przykład: minimalny automat dla (ab)* ma 3 stany, są też 3 klasy abstrakcji: eps, a, aa. Uwaga: tu ważne jest, że definiujemy automat tak, że jest funkcja przejścia jest funkcją całkowitą.

Co więcej można pokazać następujący fakt. **Pytanie:** czy wszystkie automaty o minimalnej liczbie stanów są takie same?

Fakt
Wszystkie automaty deterministyczne dla L o indeks(\sim_L) stanach są izomorficzne.

Izomorficzne, czyli można tak zmienić ich nazwy, żeby wyglądały tak samo. Tu formalnie zdefiniować izomorfizm (bijekcja t., że …). Dlaczego tak jest? Niech A_L to będzie automat minimalny rozpoznający L, taki jak zdefiniowany w dowodzie tw. Myhilla-Neroda. Niech A to jakiś inny automat rozpoznający L, taki, że jego liczba stanów jest równa liczbie stanów A_L. Chcemy pokazać, że A jest izomorficzny z A_L. Zauważmy, że wszystkie słowa, które w automacie A idą do stanu p muszą być z jednej klasy abstrakcji relacji \sim_L. Ale stanów jest |Q_L|, czyli do każdego stanu idą wszystkie słowa z pewnej klasy abstrakcji. No to już widać, że bijekcja, która przeprowadza p na p' takie, że jeśli q0 --u→ p, to p' = [u] jest izomorfizmem.

Pokazaliśmy więc, że automat minimalny jest w pewnym sensie kanoniczny. Okaże się, że możemy powiedzieć o nim więcej. Można go uzyskać z dowolnego automatu deterministycznego dla L poprzez "sklejanie" stanów. Zdefiniujemy teraz formalnie czym jest sklejenie stanów i jak to zrobić algorytmicznie.

Kongruencja

W zasadzie zawsze można skleić pewne zbiory stanów automatu i coś z tego wyjdzie. Przy czym przy niektórych sklejeniach automat może rozpoznawać zupełnie inny język. Przykład: sklejamy stan początkowy ze śmietnikiem w automacie dla (ab)*. Może też wyjść automat niedeterministyczny. Poza tym jak sklejamy stan akceptujący z nieakceptującym, to jaki ma być stan sklejony? Podamy teraz warunki na sensowne sklejenie, takie będzie nazywane kongruencją. Kongruencja to relacja równoważności R \subseteq Q x Q taka, że:

1) jeśli p i p' są w relacji R, to albo oba są akceptujące albo żaden
2) jeśli p i p' są w relacji R, p --a→ q, p' --a→ q', to również q i q' są w relacji R

Sklejenie oczywiście skleja każdą klasę abstrakcji R w jeden stan.
Automat ilorazowy A_R = (Q_R, Sigma*, q_R, F_R, delta_R) jest zdefiniowany następująco:

- stany Q_R to klasy abstrakcji relacji R, czyli {[q]_R | q \in Q}
- stan początkowy q_R to klasa abstrakcji stanu początkowego q
- stany końcowe F_R to te klasy, które zawierają stany końcowe Q
- funkcja przejścia zawiera p --a→ q jeśli przed sklejeniem z pewnym stanie była taka tranzycja (co więcej to znaczy, że w każdym stanie była taka tranzycja)

Łatwo pokazać, że L(A_R) = L(A). Bieg przed sklejeniem przenosi się na bieg po sklejeniu, a bieg po sklejeniu przenosi się na pewien bieg przed sklejeniem (tu istotnie korzystamy z własności 1) i 2)). Zauważmy też, że A_R jest deterministyczny. Czyli fajnie, dzielenie przez kongruencję (czyli sklejanie klas abstrakcji tej kongruencji) nie zmienia języka i pozostawia automat deterministyczny. Teraz pokażemy, że dla każdego automatu istnieje kongruencja, że jak przez nią podzielimy, to dostaniemy automat minimalny. A ta relacja jest zdefiniowana w naturalny sposób. **Pytanie:** może ktoś ma pomysł jak? Sklejamy stany p i q wtedy, gdy L(p, A) = L(q, A). Definiujemy: L(p, A) to jest zbiór słów, które zostaną zaakceptowane w automacie A jeśli ustalimy p na stan początkowy. Zauważmy, że to jest faktycznie kongruencja (łatwo to zobaczyć). Załóżmy dla uproszczenia, że wszystkie stany automatu A są osiągalne (jeśli nie, to trzeba najpierw wywalić te nieosiągalne). Niech n to wielkość automatu A_L. Pomyślmy ile może być stanów po sklejeniu. Nie może być mniej niż n, bo to by dało mniejszy automat dla L. Ale nie może być też więcej niż n. Bo przypuśćmy, że do tych stanów prowadzą słowa w1, ... , w(n+1). Dla każdych wi oraz wj mamy stany qi oraz qj, do których prowadzą takie, że L(pi, A) \neq L(pj, A). A więc wi \not\sim_L wj. A więc to dałoby przynajmniej n+1 klas abstrakcji relacji \sim_L, sprzeczność. A więc grup jest dokładnie n, czyli automat A_R ma dokładnie n stanów, czyli jest minimalny.

Podsumowanie

Czyli jest jeden kanoniczny automat minimalny, o najmniejszej liczbie stanów. Można go otrzymać z każdego innego poprzez sklejenie stanów, takie, że sklejamy wszystkie stany p o równym języku L(p, A).

Algorytm

Pytanie jak obliczyć ten minimalny automat. Trzeba po prostu umieć obliczyć dla których par stanów p oraz q zachodzi L(p, A) = L(q, A). Powiemy, że stany p i q są n-rozróżnialne jeśli istnieje słowo w długości co najwyżej n takie, że w \in L(p, A), ale w \not\in L(q, A), albo odwrotnie. **Pytanie**: jeśli L(p, A) i L(q, A), to jak długich słów potrzeba, żeby je rozróżnić?

Można pokazać, że n^2 wystarczy (pompowanie), ale zaraz okaże się, że nawet n wystarczy. Najpierw liczymy które stany są 0-rozróżnialne (akceptujące od nieakceptujących), a potem liczymy to dla coraz dłuższych słów. Jeśli mamy powiedziane, które pary stanów są k-rozróżnialne, to łatwo obliczyć które pary są (k+1)-rozróżnialne, po prostu w jednym kroku musimy po tej samej literze osiągnąć parę stanów k-rozróżnialnych. Jak osiągniemy punkt stały, to się zatrzymujemy. Z tego co przed chwilą powiedzieliśmy mamy, że jeśli L(p, A) \neq L(q, A), to są one n^2-rozróżnialne. A więc nasz algorytm jest poprawny. A więc są one też tak naprawdę n-rozróżnialne (dygresja), bo punkt stały zostanie osiągnięty po co najwyżej n krokach. Czyli taka minimalizacja działa w O(n) fazach po O(n) czasu, czyli O(n^2) czasu. Istnieje też lepszy algorytm (Hopcrofta), działający w czasie O(n log n). Do tej pory otwarty jest problem, czy da się to zrobić lepiej niż O(n log n), np. w czasie liniowym.

Przykład 1: automat dla (ab)* o 5 stanach. Zrobić dokładnie.
Przykład 2: automat liczący mod 3 o 9 stanach.

Wykład 5

Minimalizacja NFA
No dobrze, zminimalizowaliśmy automaty deterministyczne, ale co z minimalizacją automatów niedeterministycznych? Czy też tak się da? Niestety nie, nie ma nawet kanonicznego automatu minimalnego. Przykład: 2 automaty akceptujące język K = {ab, ac, ba, bc, ca, cb}. Możemy zrobić 2 automaty wielkości 5. Trzeba by w zasadzie pokazać, że nie da się tego zrobić automatem wielkości 4. Pokażemy to za chwilę.

Fooling set
Normalnie się tego nie robi, ale wg mnie to warto zrobić, bo jest fajne.
Tu zrobimy dygresję, jak pokazywać, że nie ma mniejszego automatu. **Pytanie**: dla deterministycznych jak pokazać, że nie ma automatu wielkości 7? Wskazać 8 słów nie w relacji M-N. Ale dla niedeterministycznego to nie wystarczy. Dlaczego? Wystarczy wziąć NFA mniejszy niż najmniejszy deterministyczny. Da się tak, np. n-ta litera od końca to a. NFA ma n+1 stanów. A DFA musi mieć przynajmniej 2^n, bo każde ze słów n literowych ma gdzieś różną literę, więc są nie w relacji M-N. Dobrze, to jak pokazywać, że NFA musi mieć przynajmniej ileś stanów? Z pomocą przychodzi następujący fakt.

Fakt (fooling set technique)
Niech L regularny. Jeśli istnieje n par słów u_1, …, u_n oraz v_1, …, v_n takich, że:
- u_i v_i \in L dla każdego i
- u_i v_k \not\in L albo u_k v_i \not\in L dla każdych i, k,

to wtedy minimalny automat niedeterministyczny dla L musi mieć przynajmniej n stanów.

Dowód

Ustalmy jakiś automat niedeterministyczny A dla języka L. Jeśli $u_i v_i \in L$, to znaczy, że musi być jakiś zbiór niepusty zbiór stanów S_i taki, że do każdego z tych stanów da się dojść po u_i ze stanu początkowego i z każdego z nich da się dojść po v_i do jakiegoś stanu końcowego.
Z drugiego warunku wynika, że $S_i \cap S_j = \emptyset$. A zatem suma po S_i musi mieć przynajmniej n stanów.
c.n.d.

Zobaczmy teraz, że faktycznie nie ma dwóch automatu wielkości 4 dla K = {ab, ac, ba, bc, ca, cb}. Niech $u_1 = eps$, $u_2 = a$, $u_3 = b$, $u_4 = c$, $u_5 = ab$ oraz $v_1 = ab$, $v_2 = b$, $v_3 = c$, $v_4 = a$, $v_5 = eps$. Mamy $u_i v_i \in L$, a dla każdych dwóch różnych indeksów i, k któreś ze słów albo jest za krótkie albo ma dwie takie same litery.
Czyli faktycznie nie ma kanonicznego najmniejszego NFA.

Problemy decyzyjne
Jest kilka naturalnych pytań, które możemy zadań odnośnie danego języka regularnego.
Oto lista tych, którym się przyjrzymy:

1) dane słowo w, automat A, czy w należy do L(A)?
2) dany automat A, czyli L(A) pusty?
3) dane automaty A, B, czy L(A) = L(B)?
4) dane automaty A, B, cz $L(A) \subseteq L(B)$?

Każde z tych pytań można zadać dla automatu deterministycznego i niedeterministycznego. Przyjrzyjmy się tym problemom. Jak zrobić 1) dla DFA? Po prostu lecimy po słowie, wychodzi w $O(|w| \, |A|)$. **Pytanie:** a jak dla NFA? Też lecimy po słowie, tylko trzymamy zbiór stanów do którego możemy dojść. Też PTIME, dokładniej $O(|w| \, |A|^2)$.
Problem 2) też łatwo w PTIME dla obu, patrzymy czy jest ścieżka z jakiegoś początkowego do jakiegoś końcowego. Jak dla 3)? Dla DFA łatwo, dopełniamy drugi, przecinamy i badamy niepustość. A dla NFA? To się okaże trudne, nie znamy algorytmu wielomianowego. Podobnie zresztą sprawdzanie, czy L(A) = Sigma*, nie znamy tu algorytmu wielomianowego i podejrzewamy, że go nie ma (problem jest PSPACE-trudny, pod koniec wykładu być może powiemy co to znaczy). Można zdeterminizować i zrobić to, co dla DFA, działa w czasie wykładniczym. Dla 4) robimy podobnie jak dla 3), bo to jest równoważne pytaniu, czy $L(A) \cap \bar{L(B)} \neq \emptyset$.

Algebraiczne podejście dla języków regularnych
Teraz zrobimy temat nieco dodatkowy, tzn. zajmiemy się algebraicznym podejściem do języków regularnych. Tego się zwykle nie robi, ale my w tym roku zrobimy. Rozważmy pewien automat deterministyczny A. Bardzo przydatnym pojęciem dla słowa $w \in Sigma^*$ jest funkcja $f_w: Q \to Q$, która mówi które stany przechodzą na które po przejściu przez słowo w. To znaczy definiujemy ją tak, że $f_w(p) = q$ wtw. $p \xrightarrow{w} q$ w automacie A. Dużo zadań można rozwiązać używając tej funkcji, a co ważniejsze można łatwiej myśleć używając tego pojęcia.

Przykład:
Niech L regularny, pokaż, że \sqrt{L} = {w | ww \in L} jest regularny.
Weźmy DFA A dla języka L. Niech A' oblicza funkcję po słowie w, jego stany to różne możliwe funkcje. f_epsilon = id, a potem łatwo liczyć. Stany akceptujące to te funkcje f, dla których f(f(q0)) \in F. Tak samo można zrobić dla \sqrt^7{L} i sporo innych podobnie.

Zobaczmy, że takie funkcje są monoidem, czyli strukturą (M, *) taką, że:

- istnieje element neutralny
- każde dwa elementy można przemnożyć i dostać znów element z dziedziny

Grupy to szczególny rodzaj monoidów, one mają jeszcze elementy odwrotne, monoidy niekoniecznie. Okazuje się, że takie spojrzenie jest bardzo przydatne. Pokażemy teraz twierdzenie, które będzie w nowy sposób charakteryzowało języki regularne. Potem wprowadzimy nowy rodzaj automatów - automaty dwukierunkowe i używając monoidów pokażemy, że one nie umieją zrobić więcej niż automaty jednokierunkowe.

Homomorfizm między dwoma monoidami (M, *1) i (N, *2) to funkcja h: M → N taka, że dla każdych m, m' in M zachodzi: h(m1) *2 h(m2) = h(m1 *1 m2).

Powiemy, że język L jest rozpoznawany przez monoid M jeśli istnieje homomorfizm h: Sigma* → M oraz podzbiór akceptujący F \subseteq M taki, że

h^{-1}(F) = L,

czyli słowa są akceptowane jeśli przechodzą na zbiór akceptujący.

Przykłady jak można rozpoznać języki regularne przez monoid:

1. Słowa długości przystającej do 3 modulo 1. Monoid: {0, 1, 2} z działaniem dodawania modulo 3. Funkcja h: Sigma* -> {0, 1, 2}, zwraca długość modulo 3. To jest homomorfizm, bo h(uv) = h(u) + h(v). Dajemy F = {1}. Mamy wtedy L = h^{-1}(F).
2. Słowa postaci a* b* c*. Monoid: {nic, a, b, c, ab, bc, zle}. Działanie na logikę. F = M \setminus {zle}. L = h^{-1}(F).
3. Słowa o podciągu abc. Monoid: {jest, nie ma} x {pref. bc, pref c, pref nie c} x {suf. ab, suf. a, suf. nie a}. F = {jest} x all x all. Działa.
4. Słowa a^n b^n. Monoid: {a długości i, b długości i, a^i b^j, b przed a}. Ale tu nieskończony.
5. Dowolny język. Monoid: (Sigma*, konkatenacja), F = L, h = id. Wtedy L = h^{-1}(F). Ale tu nieskończony.

Tw.
L jest regularny wtw. L jest rozpoznawalny przez pewien monoid **skończony**.

Dowód
Najpierw w prawo. Niech L regularny i A to automat deterministyczny, który go rozpoznaje, ze zbiorem stanów Q. Wtedy robimy M = (Q^Q, złożenie), a F to funkcje t., że stan początkowy A

przeprowadzają na jakiś akceptujący. Mamy też, że h(w) = f_w jest homomorfizmem, bo h(u) h(v) = h(uv). Poza tym h^{-1}(F) = L, co kończy dowód tej implikacji.

Teraz w lewo. Przypuśćmy, że L jest rozpoznawalny przez monoid (M, *). Automat A po prostu liczy do jakiego elementu monoidu dane słowo się wyliczy. Czyli zbiór stanów to M. Stan początkowy to h(eps). Gdy mamy h(w), to po literze a przechodzimy do h(w) \circ h(a). Stany akceptujące to te z F.
c.n.d.

Wykład 6

To jest kolejna charakteryzacja języków regularnych (już 4-ta), co pokazuje, że są rzeczywiście interesującym obiektem. Teraz zdefiniujemy automaty dwukierunkowe. One będą zachowywały się nieco inaczej niż normalne automaty. Z powodów technicznych stany będą nie między literami, ale na literach. Przykład automatu, który rozpoznaje Sigma* a Sigma^{n-1}, czyli n-ta litera od końca to a. Idea: on idzie do końca, a potem się wraca licząc do n i sprawdza, czy tam jest a. Może to nawet zrobić deterministycznie!

Formalnie A = (Sigma, |-, -|, Q, I, F, delta), gdzie
- Sigma to alfabet, |- oraz -| to specjalne symbole ograniczające słowo odpowiednio z lewej i z prawej strony
- Q to zbiór stanów
- I to stany początkowe
- F to stany akceptujące
- delta to relacja przejścia, mamy delta \subseteq Q x (Sigma \cup {|-, -|}) x Q x {-1, 0, 1}.

Interpretacja relacji przejścia jest jak zwykle, przy czym {-1, 0, 1} oznacza w którą stronę automat teraz pójdzie. Zaczynamy zawsze w stanie początkowym na pierwszej literze. Kończymy w stanie końcowym.

Przykład:
1. Napiszmy automat dla języka: n-ta litera od końca to a.
   Q = {p, q1, …, qn, akc, rej}
   A = (Sigma, |-, -|, Q, p, {akc}, delta)
   Do delta należą:
   (p, wszystko oprócz -|, p, 1) - idziemy w prawo
   (p, -|, q1, -1)
   (qi, wszystko, q(i+1), -1)
   (qn, a, akc, 0)
   (qn, nie a, rej, 0)
   (rej, wszystko, rej, 0)
2. Każdy język regularny da się zrobić. Po prostu stan, który powinien być przed literą stawiamy na literze. Stan końcowy odczytujemy na literze -|.

Teraz pytanie brzmi, czy da się zrobić coś więcej niż języki regularne? Nie, nie da się, już wcześniej mówiłem. Można to łatwo pokazać używając monoidów.

Tw.
Automaty dwukierunkowe rozpoznają dokładnie języki regularne.

Dowód
To, że akceptują języki regularne, to już pokazaliśmy powyżej.
Teraz rozważmy pewien automat dwukierunkowy A. Najpierw załóżmy dla uproszczenia, że A jest deterministyczny, czyli, że ma dokładnie jedno przejście dla ustalonego stanu i litery. Pokażemy, że jego język da się rozpoznać pewnym monoidem skończonym, co zakończy dowód. Zawsze jak projektujemy taki monoid musimy się zastanowić co właściwie chcemy wiedzieć o słowie, czego nam potrzeba. Wtedy monoid będzie zbiorem wszystkich możliwych takich informacji, a słowo będzie mapowane przez homomorfizm na odpowiednią wartość. Tutaj żeby wiedzieć wszystko o słowie wystarczy wiedzieć co się stanie jeśli automat wejdzie do niego z lewej (lub z prawej) w danym stanie - czy wyjdzie z lewej lub z prawej, zaakceptuje lub się zapętli. A więc dla słowa w zdefiniujmy funkcję, która mówi, że jeśli zaczynamy w słowie na pierwszej pozycji z lewej/prawej w danym stanie, to jakie będzie możliwe wyniki: wyjście z lewej/prawej w danym stanie, akceptacja, zapętlenie się w środku. Czyli f_w: Q x {L, P} -> Q x {L, P} \cup {akc, pętla}. Takie funkcje można mnożyć tak, żeby wyszedł homomorfizm, bo da się odtworzyć co funkcja robi na słowie uv, gdy wiemy co robi na u i co robi na v. A więc monoid to (funkcje, operacja na nich). Funkcje akceptujące to te, dla których jak zaczynamy na pierwszej pozycji z lewej w stanie początkowym, to zaakceptujemy. A więc pokazaliśmy, że deterministyczne automaty dwukierunkowe akceptują tylko języki regularne. Analogicznie można zrobić dla niedeterministycznych, tylko, że trzeba pamiętać relację, czyli wszystkie możliwe przejścia. Wtedy zbiór akceptujący to taki, że istnieje zaczęcie ze zbioru stanów początkowych takie, że to doprowadzi do akceptacji.
c.n.d.

Problemy otwarte dotyczące automatów dwukierunkowych
Jak duży jest wzrost gdy przekształcamy automat niedeterministyczny (1 lub 2 kierunkowy) na automat deterministyczny 2 kierunkowy?

Gramatyki bezkontekstowe
Zaczynamy teraz drugi duży dział na tym przedmiocie, dotyczący języków bezkontekstowych. Mogą być one opisywane albo przez gramatyki bezkontekstowe albo przez automaty ze stosem. Zaczniemy od gramatyk bezkontekstowych, bo one są prostsze do zrozumienia.

Języki bezkontekstowe to większa klasa niż regularne. Okaże się, że każdy język regularny jest bezkontekstowy, ale pewne języki bezkontekstowe nie są regularne.

Zaczniemy od przykładu, palindromy nad dowolnym skończonym alfabetem (np. {a, b}) są językiem bezkontekstowym. Gramatyka dla nich to:

P -> aPa
P -> bPb
P -> a
P -> b
P -> eps

Idea jest taka, że zaczynamy od symbolu P, a potem aplikujemy reguły. Język to zbiór rzeczy, które możemy dostać. Np. możemy uzyskać słowo abbba w następujący sposób:

P -> aPa -> abPba -> abbba

Definicja
Teraz formalna definicja. Gramatyka (T, N, S, delta) składa się ze:
- zbioru terminali T (symboli końcowych, czyli literek)
- zbioru nieterminali N (symboli pomocniczych, czasem mówimy zmiennych, dużych liter)
- symbolu startowego S \in N
- zbioru produkcji delta \subseteq N x (N \cup T)*

Powiedzieć co jest czym w gramatyce powyżej i że tam był tylko jeden nieterminal, ale w ogólności może być więcej.

Produkcje (czasem mówimy reguły, albo też może inaczej) zapisujemy często jako X --> alpha zamiast (X, alpha) \in delta. Reguły rozszerzamy też do relacji na ciągach nieterminali,
tzn. np. gdy X → alpha, to piszemy również beta X gamma → beta alpha gamma.

Wyprowadzenie (derywacja) to ciąg
alpha_0 → … → alpha_n
taki, że alpha_i → alpha_{i+1}, alpha_0 = S oraz alpha_n \in T*.
Wówczas mówimy, że alpha_n da się wyprowadzić w gramatyce G.
Język L(G) gramatyki G to zbiór wszystkich słów, które dadzą się wyprowadzić w gramatyce G.

Czasem wyprowadzeniem nazywamy też wyprowadzenie bez założeń alpha_0 = S i alpha_n \in T*, nie będziemy wprowadzać dodatkowych definicji.

Przykłady
- a^n b^n: P → aPb | eps
- {w | #a(w) = #b(w)}: P → aPb | bPa | PP | eps
  (tu dowód potrzebny - robimy wykres #a(w) - #b(w), patrzymy gdzie przecina 0)
- wyrażenia nawiasowe P → PP | (P) | eps
- a^n b^k, n \neq k P → A | B, A → aA | aR, B → Bb | Rb, R → aRb | eps
- wyrażenia arytmetyczne W → W + W | W * W | (W) | 0 | 1

- palindromy długości podzielnej przez 3 — X0 → a X1 a | b X1 b | eps, X1 → a X2 a | b X2 b | a | b, X2 → a X0 a | b X0 b

Bezkontekstowe vs regularne

Fakt
Każdy język regularny jest bezkontekstowy

Dowód
Najpierw przykład. Weźmy automat rozpoznający (ab)*c(b+c)*. Ma 3 stany, nie licząc śmietnika, bo go tu nie potrzebujemy: q0, q1, q2, stan q0 początkowy, stan q2 akceptujący. Ma tranzycje q0 --a→ q1, q1--b→ q0, q0 --c→ q2, q2 --b→ q2, q2 --c→ q2.

Robimy gramatykę. Terminale {a, b, c}, nieterminale: X0, X1, X2. Początkowy nieterminal X0.
Produkcje:
X0 → a X1
X1 → b X0
X0 → c X2
X2 → b X2
X2 → c X2

X2 → eps
Te wcześniejsze odpowiadają tranzycjom automatu, ta późniejsza faktowi, że q2 jest akceptujacy.

Zobaczmy jak wyprowadzić abcbb.
X0 → a X1 → a b X0 → a b c X2 → a b c b X2 → a b c b b X2 → a b c b b
Widać, że to odpowiada biegowi automatu.

W ogólności dla automatu (Q, Sigma, q0, F, delta) robimy
terminale = Sigma, nieterminale = X_q, q \in Q (albo Q, ale to by się myliło), nieterminal początkowy to X_q0, dla każdej tranzycji p --a→ q robimy produkcję X_p → a X_q oraz dla każdego stanu końcowego p robimy produkcję X_p → eps.
c.n.d.

Drzewa wyprowadzeń
Przykłady:
- dla gramatyki X → a X b | b X a | X X | eps oraz słowa aababbba
- dla gramatyki palindromów długości podzielnej przez 3 oraz słowa abbaaabba

Pytania:

- czy dla każdej derywacji istnieje odpowiadające drzewo derywacji? (TAK)
- czy dokładnie jedno? (TAK)
- czy dla każdego drzewa derywacji istnieje odpowiadająca derywacja? (TAK)
- czy dokładnie jedna? (NIE)

Ale można dla każdego drzewa derywacji wyszczególnić jedną derywację, tzw. lewostronną derywację. To odpowiada stosowaniu zawsze reguły dla najbardziej lewego nieterminala.

Wykład 7

Lemat o pompowaniu

Podobnie jak przy lemacie o pompowaniu dla języków regularnych łatwiej jest tu zapamiętać dowód niż sformułowanie. Ja np. nie pamiętam sformułowania, tylko wyprowadzam je pamiętając ideę dowodu.

Dla każdego języka bezkontekstowego L istnieje taka stała N, że dla każdego słowa w \in L długości co najmniej N istnieje podział w = x s y t z taki, że:

- st \neq eps
- |syt| \leq N
- dla każdego i zachodzi x s^i y t^i z \in L

Dowód

Ustalmy gramatykę bezkontekstową G taką, że L = L(G). Niech n to liczba nieterminali w gramatyce i niech k to maksymalna liczba symboli po prawej stronie produkcji.
Zauważmy, że jeśli mamy drzewo wyprowadzenia o głębokości nie większej niż n, to słowo z niego wyprowadzone ma długość co najwyżej n^k. Ustalmy N = n^{k+1} + 1. Niech w \in L takie, że |w| >= N. Wówczas drzewo wyprowadzenia dla w na głębokość co najmniej n+2, czyli jest pewna ścieżka, która ma długość co najmniej n+2. Na tej ścieżce jest więc co najmniej n+1 nieterminali, czyli któryś powtarza się co najmniej 2 razy. Niech będzie to X, pod nim jest drugi X. A więc mamy wyprowadzenie X → s X t na st \neq eps. Tu narysować ogólną sytuację. Jak wstawimy (lub usuniemy) takie drzewo wyprowadzenia do tego co mamy to dostaniemy x s^i y t^i z \in L. Dodatkowo możemy wziąć najniższą taką parę, wtedy dostaniemy |syt| <= n^{k+1}, czyli |syt| <= N.
c.n.d.

Ten lemat o pompowaniu, podobnie jak lemat o pompowaniu dla języków regularnych nie zawsze działa. Z tego co wiem, to nie ładnego warunku, który zawsze działa. Ale są lepsze lematy niż lemat o pompowaniu, działające na podobnej zasadzie. Zrobimy taki jeden, lemat Ogdena. Tam po prostu pompuje się uważniej.

Lemat Ogdena

Dla każdego języka bezkontekstowego L istnieje taka stała N, że dla każdego słowa w \in L z co najmniej N zaznaczonymi pozycjami istnieje podział w = x s y t z taki, że:

- st na przynajmniej jedną zaznaczoną pozycję
- syt ma mniej niż N zaznaczonych pozycji
- dla każdego i zachodzi x s^i y t^i z \in L

Dowód jest w zasadzie taki sam. Patrzymy na ścieżki idące od zaznaczonych pozycji do korzenia i drzewo stworzone przez te ścieżki, wierzchołki drzewa dajemy tam, gdzie schodzą się ścieżki. To drzewo ma stopień co najwyżej k, więc dla odpowiednio dużego N, tak jak dla zwykłego lematu o pompowaniu, istnieje pewna ścieżka na której powtarza się symbol. Tam pompujemy, tak jak wcześniej.
Zauważmy, że dla wszystkich pozycji zaznaczonych lemat Ogdena jest równoważny lematowi o pompowaniu.

Przykłady

1. Rozważmy język L = {a^n b^n c^n | n \in N}. Weźmy stałą K z lematu. Rozważmy a^K b^K c^K. Ani s ani t nie może zachodzić na obszary z różnych liter. Zatem s i t zawierają takie same litery. Zatem x s^2 y t^2 z będzie zawierać więcej niektórych liter (bo zwiększymy liczbę tylko co najwyżej dwóch liter).
2. Rozważmy język L = {a^i b^j c^k d^l | j = k = l lub i = 0}. Lemat o pompowaniu nie działa, bo zawsze może się okazać, że pompujemy litery a - i wtedy jest to nieszkodliwe, słowo nadal jest w języku. Ale z lematu Ogdena da się to zrobić. Zaznaczamy wszystko oprócz liter a i wtedy jest ok, tak samo jak w przykładzie pierwszym robimy.

Jednoznaczność
Pytanie na ile wyprowadzenia są unikalne dla słowa.
Można to pytania zadań na różnych poziomach:

- jedno wyprowadzenie dla każdego słowa (łatwo pokazać, że nie zawsze jest)
  X → XX → abX → abab, X → XX → Xab → abab
- no ale te wyprowadzenia odpowiadają jednemu drzewu derywacji, może prawdą jest, że zawsze jest jedno drzewo derywacji w danej gramatyce (łatwo pokazać, że nie zawsze jest)
  X → aX | Xa | eps, X → aX → a oraz X → Xa → a
- no dobrze, ale widać, że tu jest inna gramatyka, która byłaby jednoznaczna: X → aX | eps. Dla każdego języka bezkontekstowego jest gramatyka, która nie jest jednoznaczna, wystarczy z nieterminala X zrobić dwie wersje: X1 oraz X2. Ale ciekawym pytaniem jest: czy dla każdego języka bezkontekstowego jest pewna gramatyka, że tam jest dla każdego słowa jedno drzewo derywacji. **Pytanie**: czy zawsze istnieje taka gramatyka? Odpowiedź brzmi: NIE. Przykład języka to {a^i b^n c^n} \cup {a^n b^n c^i}. Słowo a^n b^n c^n powinno mieć zawsze dwie derywacje. Bez dowodu.

Postacie normalne
Okazuje się, że każdą gramatykę można sprowadzić do postaci, które są prostsze w obsłudze. Najbardziej znaną taką postacią jest postać normalna Chomsky'ego.

Gramatyka bezkontekstowa jest w postaci normalnej Chomsky'ego gdy występują tam tylko reguły następującej postaci:
X → Y Z, gdzie Y i Z to nieterminale
X → a, gdzie a to terminal
S → eps

Fakt
Dla każdego języka bezkontekstowego L istnieje gramatyka bezkontekstowa w postaci Chomsky'ego G taka, że L(G) = L.
Dowód
Jeśli eps \in L, to dodajemy regułę S → eps. Będziemy teraz starali się zdefiniować gramatykę dla języka L \setminus {eps}, czyli zakładamy, że eps nie należy do L.

Będziemy po kolei przekształcać naszą pierwotną gramatykę, tak, by na końcu zostały jedynie reguły postaci X → Y Z, X → a oraz X → eps.

Krok 1. Najpierw dla każdego terminala a wprowadzamy specjalny nieterminal A i dodajemy regułę A → a. Następnie każde wystąpienie litery a po prawej stronie (oprócz tej właśnie reguły) zastępujemy przez A. Po tym ruchu wszystkie reguły są albo postaci X → eps, X → a albo postaci X → Y1 … Yk. Problem polega na tym, że wciąż może być k = 1 lub k > 2.

Krok 2. Eliminacja reguł postaci X → eps.
Patrzymy z których zmiennych możemy wyprodukować słowo puste, niech będzie to zbiór P. Następnie dla każdej reguły X → alpha dodajemy reguły postaci X → beta dla każdego niepustego beta, które powstało z alpha poprzez niektórych zmiennych ze zbioru P. Następnie kasujemy wszystkie reguły X → eps. Nie zmieniliśmy przez to języka, to po prostu robimy taką derywację, że nigdy nie produkujemy zmiennych, które miałyby potem wyprodukować słowo puste.

Krok 3. Eliminacja reguł postaci X → Y.
Patrzymy które zmienne mogą przejść na które ciągiem takich kroków.
Jeśli mamy Y1 → Y2 → … → Yk, to mówimy, że Y1 może przejść na Yk.
Dla każdej pary (X, Y) takiej, że X może przejść na Y oraz dla każdej reguły Y → alpha dodajemy również regułę X → alpha. Widać, że to nie zwiększa mocy. Następnie eliminujemy wszystkie przejścia X → Y. To nie zmienia języka, to dla każdej derywacji w starej formie, która używałaby Y1 → …. → Yk → alpha robimy od razu Y1 → alpha.

Krok 3. Eliminacja reguł postaci X → Y1 … Yk dla k > 2.
Dla każdej reguły X → Y1 … Yk dla k > 2 robimy nowe zmienne Z1, …, Z(k-2) i regułę X → Y1 Z1, dla każdego i < k-2 regułę Zi → Y(i+1) Z(i+1) oraz regułę Z(k-2) → Y(k-1) Yk.
Potem usuwamy oryginalną regułę X → Y1 … Yk.
c.n.d.

Postać normalna Greibach (bez dowodu)
Można też każdą gramatykę G taką, że L(G) nie zawiera słowa pustego zamienić na gramatykę z regułami tylko postaci:
X → a Y1 … Yk dla k >= 0.
Nie podajemy dowodu, jest trochę techniczny.

Automaty ze stosem
Najpierw przykład: dla palindromów
Tranzycje to:
(q, S) --a→ (q, AS)
(q, S) --b→ (q, BS)
(q, X) → (p, X), X dowolne
(p, A) --a→ (p, eps)
(p, B) --b→ (p, eps)
(p, S) → (f, S)
Stan q jest początkowy, a f akceptujący. Startujemy z konfiguracji (q, S).

Automat (niedeterministyczny) ze stosem A = (Sigma, Q, q0, F, Gamma, s0, delta) składa się z:
- skończonego alfabetu Sigma
- zbioru stanów Q
- stanu początkowego q0 \in Q
- zbioru stanów końcowych F \subseteq Q
- alfabetu stosowego Gamma (zbioru symboli, które mogą pojawiać się na stosie)
- symbolu początkowego s0 \in Gamma
- zbioru tranzycji delta \subseteq Q x Gamma x (Sigma \cup eps) x Q x Gamma*

Konfiguracja automatu ze stosem to stan z Q oraz ciąg symboli stosowych z Gamma*.
Automat startuje w konfiguracji (q, s0). Następnie zmienia stan i czubek stosu wedle tranzycji.
Kończy w konfiguracji (f, alpha) o ile f \in F, alpha może być dowolna.
Interpretacja tranzycji (q, A, a, q', alpha) jest następująca: w stanie q, o ile na czubku stosu jest symbol stosowy A, to możemy wczytać z wejścia literę a, zmienić stan na q', a na stosie zamiast A umieścić ciąg symboli stosowych alpha. Będziemy tę tranzycję zapisywać jako
(q, A) --a→ (q', alpha). Jeśli c0 --a1→ c1 … c(n-1) --an→ cn, to piszemy c0 --w→ cn dla w = a1...an. Jeśli (q, s0) --w→ (f, alpha) dla pewnego f \in F, to mówimy, że A akceptuje w.
Język automatu, L(A), to zbiór wszystkich słów w, które A akceptuje.

Wykład 8

Przykład
Robimy automat dla języka palindromów nad {a, b} o długości podzielnej przez 3.
Sigma = {a, b}
Q = {p0, p1, p2, q0, q1, q2}
początkowy - p0
F = {q0}
Gamma = {P, A, B}
s0 = P
Idea jest taka, że w stanach p_i będziemy w pierwszej połowie słowa, a w q_i w drugiej.
Dodatkowo indeks liczy modulo 3. Czubek stosu jest z lewej.
Tranzycje to:
(p_i, X) --a→ (p_j, AX) o ile j = i+1 mod 3, X \in {A, B, P}
(p_i, X) --b→ (p_j, BX) o ile j = i+1 mod 3, X \in {A, B, P}

(p_i, X) --x→ (q_j, X) o ile j = i+1 mod 3, x \in {a, b}, X \in {A, B, P}
(p_i, X) --eps→ (q_i, X) o ile X \in {A, B, P}

(q_i, A) --a→ (q_j, eps) o ile j = i+1 mod 3
(q_i, B) --b→ (q_j, eps) o ile j = i+1 mod 3

(q_0, P) --eps→ (f, eps)

Przykładowy bieg:
(p0, P) --a→ (p1, AP) --b→ (p2, BAP) --b→ (p0, BBAP) → (q0, BBAP) --b→ (q1, BAP) --b→ (q2, AP) --a→ (q0, P) → (f, eps)
Zatem abbbba należy do języka.

Zauważmy, że nasz automat ma kilka własności:
- każda tranzycja albo wrzuca jeden symbol na stos (push) albo zdejmuje ze stosu (pop),
- jest tylko jeden stan końcowy
- każde obliczenie akceptujące kończy się usunięciem symbolu początkowego i zaakceptowaniu pustym stosem
- symbol początkowy s0 nigdy nie jest wrzucany na stos, a zawsze jest usuwany tuż przed zaakceptowaniem

Jest to ogólniejszy fakt. Każdy język akceptowany przez pewien automat można zaakceptować takim właśnie automatem.

Warianty automatów ze stosem

Tw.
Dla każdego automatu ze stosem A istnieje automat ze stosem A' taki, że L(A) = L(A') oraz dodatkowo A' spełnia warunki:

- każda tranzycja jest albo typu pop albo typu push
- jest tylko jeden stan końcowy
- każde obliczenie akceptujące akceptuje pustym stosem
- symbol początkowy nigdy nie może być wrzucony na stos (nie ma takiej tranzycji) i zawsze jest usuwany tuż przed akceptacją

Uwaga: zauważmy, że w takim razie automat A' akceptuje wtw. gdy jest stos jest pusty. Można zatem zmienić jego warunek akceptacji mówiąc, że akceptuje, gdy stos jest pusty. Będziemy więc od tej pory mogli rozważać dwa alternatywne warunki akceptacji:

- przez stan
- przez pusty stos

Zobaczmy, że akceptacja przez pusty stos nie jest mocniejsza, bo możemy dla każdego automatu, który akceptuje przez pusty stos wrzucić coś na dół i jak to zobaczymy na dole to przejść do specjalnego stanu akceptującego (nowego).
Można by teoretycznie rozważać też warunek przez stan oraz pusty stos, ale to nie jest wcale wygodniejsze.

Dowód (twierdzenia)
Dodajemy nowy stan początkowy q0 i symbol początkowy P0, robimy też tranzycję (q0, P0) → (q0', P0' P0), gdzie q0' to stary stan początkowy, a P0' to stary symbol początkowy.
Robimy też nowy stany końcowy f i normalny q oraz tranzycje

(f', X) → (q, X) dla każdego starego stanu akceptującego f' i zmiennej X
(q, X) → (q, eps) dla każdej zmiennej X różnej od P0
(q, P0) → (f, eps)

One pozwalają na opróżnienie stosu przed akceptacją.
Teraz pozostaje jedynie pokazać, że wystarczy ograniczyć się do tranzycji typu push i pop.
Weźmy tranzycję (p, X) --a→ (q, Yk … Y1). Jeśli k = 0, to jest typu pop, w innym wypadku będziemy ją przerabiać. Dodajemy nowe stany r1, ... , rk oraz tranzycje:

(p, X) --a→ (r1, eps)
(r1, Y) → (r2, Y1 Y) dla dowolnego Y
(r_i, Y_{i-1}) → (r_{i+1}, Y_i Y_{i-1}) dla i < k
(r_k, Y_{k-1}) → (q, Y_k Y_{k-1})

Przerobiliśmy więc dowolną tranzycję na serię popów (jednego) i pushy. Stany r1, …, rk muszą być różne dla różnych tranzycji.
c.n.d.

Równoważność automatów ze stosem i gramatyk
Pokażemy teraz, że automaty ze stosem i gramatyki akceptują te same języki.

Tw.
Język L jest rozpoznawany przez pewną gramatykę bezkontekstową wtw. jest akceptowany przez pewien automat ze stosem.

Dowód
Najpierw pokażemy implikację “=>”, która jest łatwiejsza. Niech L będzie rozpoznawany przez pewną gramatykę bezkontekstową G. Możemy bez straty ogólności założyć, że G jest w postaci normalnej Chomsky’ego. Czyli ma reguły postaci S → eps (potencjalnie), X → a, X → YZ.
Robimy więc automat ze stosem A taki, że L(A) = L, będziemy zakładać, że akceptuje on przez pusty stos.
Stan jest jeden: Q = {q}. Jest on zarówno początkowy jak i akceptujący.
Symbol początkowy to S. Tranzycje są następujące. Po pierwsze mamy tranzycję (q, S) → (q, eps), o ile taka jest w gramatyce. Po drugie dla każdej reguły X → a mamy tranzycję (q, X) --a→ (q, eps).
Po trzecie dla każdej reguły X → Y Z mamy tranzycję (q, X) → (q, YZ). Widać, że język jest taki sam.

Teraz pokażemy trudniejszą implikację “<=”. Niech L będzie rozpoznawany przez pewien automat ze stosem (Sigma, Q, q0, F, Gamma, s0, delta) akceptujący przez pusty stos.
Zaprojektujemy gramatykę, która będzie rozpoznawała ten sam język. Zbiór nieterminali to {X_{p, q}^s | p, q \in Q, s \in Gamma} oraz specjalny symbol S. Idea jest taka, że język L(X_{p, q}^s) będzie równy
{w | (p, s) --w→ (q, eps)}.
Zbiór terminali to oczywiście Sigma. Natomiast nieterminal początkowy to S, dla którego mamy reguły: S → X_{q0, f}^s0 dla każdego f \in F. Oprócz tego mamy następujące reguły.
Dla każdej tranzycji pop (p, Y) --a→ (q, eps) robimy produkcję X_{p,q}^Y → a. Natomiast dla każdej tranzycji (p, Y) --a→ (q, ZY) robimy produkcje X_{p,r}^Y → a X_{q,s}^Z X_{s,r}^Y dla dowolnych stanów r, s. Wystarczy zobaczyć, że ta gramatyka rzeczywiście generuje język L.

Zauważmy najpierw, że wszystko z języka może być rzeczywiście wygenerowane przez gramatykę. Patrzymy na bieg i aplikujemy odpowiednią tranzycję (na rysunku łatwiej zobaczyć).
Z drugiej strony niech jakieś słowo będzie wygenerowane przez gramatykę. Zobaczmy na drzewo derywacji tego słowa w gramatyce. Patrząc na to drzewo od dołu budujemy kawałki biegów, które się dobrze składają. W korzeniu dostajemy bieg automatu ze stosem taki jak powinien być.
c.n.d.

Własności domknięcia
Zobaczymy jakie są własności domknięcia języków bezkontekstowych.
Po pierwsze są one zamknięte na **sumę**. Niech L1 = L(G1) i L2 = L(G2). Łatwo możemy zrobić G taką, że L(G) = L1 \cup L2. Po prostu robimy S → S1, S → S2, a resztę tak jak w gramatykach G1 i G2.

Po drugie **nie** są one zamknięte na **przecięcie**. Łatwo pokazać, że {a^n b^n c^k | n, k \in N} oraz {a^k b^n c^n | n, k \in N} są bezkontekstowe. Natomiast ich przecięcie to {a^n b^n c^n | n \in N} co jak wiemy nie jest bezkontekstowe. Z tego wynika też, że języki bezkontekstowe **nie** są zamknięte na **dopełnienie**.

Warto natomiast zauważyć, że języki bezkontekstowe są zamknięte na **przecięcie z językami regularnymi**. Po prostu robimy produkt automatu ze stosem rozpoznającego dany język bezkontekstowy z automatem skończonym rozpoznającym dany język regularny. Jako wynik dostajemy znów automat ze stosem.

Deterministyczne automaty ze stosem
Dla automatów skończonych automaty deterministyczne i niedeterministyczne definiują tę samą klasę. **Pytanie** do sali: czy dla automatów ze stosem będzie tak samo? Nie, nie będzie tak samo. Trzeba najpierw powiedzieć dokładnie co to znaczy, że automat ze stosem jest deterministyczny. Bo teraz mamy też epsilony, musimy powiedzieć co znaczy determinizm przy ich obecności. Mówimy, że automat jest deterministyczny jeśli z danej konfiguracji albo da się wyjść epsilonem, ale niczym innym albo da się wyjść po każdej literze na tylko jeden sposób.

Okazuje się w szczególności, że deterministyczne automaty ze stosem są zamknięte na dopełnienie.

Tw.
Jeśli L jest rozpoznawalny przez deterministyczny automat bezkontekstowy, to dopełnienie L też.

Dowód (idea)
Gdyby nie było epsilonów to łatwo - po prostu zmieniamy stany akceptujące na nieakceptujące. Tu musimy trochę popracować z epsilonami.

Najpierw modyfikujemy nasz automat tak, żeby spełniał 3 warunki:
- symbol początkowy nie jest nigdy zdejmowany ani wkładany na stos
- zawsze jest przejście albo po epsilonie albo po każdej literze (i nic więcej)
- zawsze da się przeczytać całe słowo (nie ma pętli po epsilonie)

To się da dość łatwo zrobić (dodajemy nowy symbol, dodajemy brakujące przejścia, które nic nie robią, wykrywamy pętle po epsilonie). Teraz robimy automat dla dopełnienia. Robimy Q' = Q x {0, 1, 2}. Druga współrzędna pamięta, czy dla danego słowa był już stan akceptujący, czy nie. Wartość 0 mówi, że stanu akceptującego nie było i nie będzie (bo zamierzamy właśnie przeczytać literę), wartość 1, że był stan akceptujący, a wartość 2, że nie było, ale czytamy jeszcze epsilony. Dodajemy opcję przejścia z 2 do 0 jeśli widzimy, że nie da się dalej iść epsilonami. Stan początkowy to (q0, 2), a końcowy to (f, 0). To akceptuje dopełnienie.
c.n.d.

Własności domknięcia dla deterministycznych języków bezkontekstowych

Jak widzimy są zamknięte na dopełnienie.

Nie są też zamknięte na przecięcie, bo {a^i b^i c^j} przecięte z {a^i b^j c^j} daje {a^i b^i c^i}, które nie jest bezkontekstowe. Natomiast dwa poprzednie są.
W związku z tym nie są też zamknięte na sumę, bo inaczej byłyby na przecięcie też (bo na dopełnienie są).

Wykład 9

Problemy decyzyjne
Co chcemy wiedzieć o danej klasie języków? Chcemy znać:
- różne sposoby definiowania (tu gramatyki i automaty ze stosem)
- kiedy język jest w danej klasie (lemat o pompowaniu, konstruowania gramatyki lub automatu)
- jakie są własności domknięcia (znamy podstawowe)
- podstawowe problemy decyzyjne (teraz zrobimy)

Dla regularnych pytaliśmy podobnie, teraz zapytamy o:
- czy słowo w należy do języka L
- czy język L pusty
- czy dwa języki K i L są równe
- czy K zawiera się w L

Dla pierwszego problemu pokażemy zaraz algorytm, tzn. algorytm CYK (Cocke, Younger, Kasami), będzie działał w O(n^3) (da się szybciej, w czasie O(n^omega), gdzie jest to najlepszy czas mnożenia dwóch macierzy).

Czy L jest pusty? Po prostu trzeba to sprawdzić na gramatyce. Patrzymy które zmienne mogą wyprodukować literę i patrzymy, czy początkowy symbol może przejść na taką zmienną. W czasie wielomianowym (być może liniowym).
Teraz okazuje się, że równość dwóch języków bezkontekstowych jest problemem nierozstrzygalnym, czyli nie ma dla niego algorytmu. Tym bardziej zawieranie. Wrócimy do tego przy okazji maszyn Turinga, bo wtedy będziemy umieli precyzyjnie dowodzić takie rzeczy.

Co ciekawe Senizergues w 1997 roku pokazał, dla dwóch języków akceptowanych przez deterministyczne automaty ze stosem równość jest rozstrzygalna (istnieje algorytm). To miało wiele stron (około 100 chyba). Teraz jest lepszy dowód Petra Jancara (może 2012 rok), około 15 stron, ale wciąż mocno skomplikowany. I nadal nie jest znana żadna rozsądna złożoność podczas gdy nie jest wykluczone (z tego co wiem), że problem jest rozwiązywalny w czasie wielomianowym.

Algorytm CYK
Mamy daną gramatykę G oraz słowo w, chcemy sprawdzić, czy w należy do języka L(G).

Pomysł jest taki, żeby użyć programowania dynamicznego i sprawdzać dla coraz dłuższych infiksów słowa, czy ten infiks da się wyprodukować z danego terminala.
Zrobimy więc tabelę trójwymiarową T[i][j][X], która będzie zawierała 1 o ile słowo w[i..j] należy do L(X) (czyli da się wyprodukować z nieterminala X), a 0 w przeciwnym przypadku.
Chcemy obliczyć T[1][n][S], gdzie n to długość w, a S to symbol startowy.
Zakładamy, że G jest w postaci normalnej Chomsky'ego (jeśli nie, to możemy ją do takiej przerobić). A więc najpierw liczymy T[i][i][X]. To jest prawda o ile tylko mamy w gramatyce produkcję X → a, gdzie a = w[i]. Potem liczymy T[i][j][X] dla coraz większych j-i.
Po prostu T[i][j][X] jeśli istnieje produkcja X → YZ oraz indeks k taki, że i < k <= j oraz T[i][k-1][Y] = T[k][j][Z] = 1. To się da obliczyć w czasie O(j-i). A zatem cały algorytm działa w czasie O(|G| n^3), bo każdą z |G| n^2 komórek da się wypełnić w czasie O(n).

Uwaga: da się to zrobić w czasie O(|G| n^omega), (może razy polylog(n)) gdzie omega to minimalna liczba t., że mnożenie macierzy n x n da się zrobić w czasie n^omega. Widać, że liczenie T[i][j][X] ma trochę wspólnego z mnożeniem macierzy.

Tw. Parikha
Twierdzenie Parikha to przydatny fakt o językach bezkontekstowych. Pokazuje intuicyjnie jakie języki bezkontekstowe możemy wygenerować jeśli nie interesuje nas kolejność liter, a jedynie to ile ich było. Bardzo ładny jest jego dowód i robię je tu również z tego powodu. Pokazuje, że żeby coś udowodnić o językach bezkontekstowych warto patrzeć na drzewa derywacji (podobnie jak to robiliśmy przy okazji lematu o pompowaniu) i jak to robić.

Najpierw kilka definicji. Niech Sigma = {a1, …, ak}. Dla słowa w in Sigma* definiujemy jego obraz Parikha, który jest wektorem w N^k: PI(w) = (#a1(w), …, #ak(w)). PI do Parikh image. Ten wektor po prostu mówi ile jest których liter w słowie w. Obraz Parikha języka L to po prostu PI(L) = {PI(w) | w in L}, czyli zbiór wektorów, które można uzyskać słowami z języka.

Twierdzenie Parikha będzie mówiło jakie obrazy Parikha mają języki bezkontekstowe.

Zbiór S \subseteq N^k jest **liniowy** jeśli jest postaci {b + n1 v1 + … + nd vd | ni in N}, gdzie b, v1, …, vd in N^k. Zbiór S \subseteq N^k jest **semiliniowy** jeśli jest skończoną sumą zbiorów liniowych. Przykłady to:

- {(0,0) + n(1,1) | n in N} - przekątna
- {(0,0) + n(1,2) + m(2,1) | n, m in N} - skrzywiona krata
- {(10,10) + n(5,0) + m(0,5) + k(3,3) | n, m, k in N} - to jest taki dziwny zbiór
- suma wszystkich powyższych

Tw. Parikha
Obraz Parikha języka bezkontekstowego jest semiliniowy.

Przykłady:

1. {a^n b^n | n in N} ma obraz Parikha {n(1, 1) | n in N}
2. {w | #a(w) = #b(w)} ma taki sam obraz Parikha
3. {a^i b^j c^k | j = i+k} ma obraz Parikha {k(1, 1, 0) + n(0, 1, 1) | k, n in N}
4. L(a*(bc)*(a+c)*) ma obraz Parikha {k(1, 0, 0) + n(0, 1, 1) + m(0, 0, 1) | k, n, m in N}

Uwaga: Język, który nie jest bezkontekstowy też może mieć semiliniowy obraz Parikha, np. {a^n b^n c^n | n in N} oraz (abc)* mają ten sam obraz Parikha.

Dowód
Zakładamy, że mamy gramatykę w postaci Chomsky'ego.
Niech **pompa** to (prawie) drzewo derywacji z korzeniem o symbolu X, wszystkimi liśćmi z terminalami, ale jednym liściem z symbolem X. Jak mamy drzewo derywacji o symbolu X gdzieś w drzewie, to możemy je taką pompą **napompować**. Podobnie możemy pompować pompy. **Alfabet** ALF(p) pompy p to zbiór wszystkich nieterminali w pompie p. Pompa jest **minimalna** jeśli nie da się jej uzyskać z mniejszej pompy o tym samym alfabecie przez napompowanie. Podobnie drzewo jest **minimalne** jeśli nie da się go uzyskać z mniejszego drzewa o tym samym alfabecie nieterminali. Zauważmy, że każde drzewo lub pompa minimalna ma głębokość nie większą niż n^2 (albo coś w tym stylu), bo większe drzewo można odpompować. A zatem jest skończenie wiele minimalnych drzew i minimalnych pomp. Teraz twierdzę, że każde drzewo derywacji można uzyskać z minimalnego drzewa derywacji o tym samym alfabecie przez pompowanie minimalnymi pompami (o nie większym alfabecie nieterminali). To w zasadzie widać. Co więcej jeśli pompujemy drzewa minimalne minimalnymi pompami, to osiągamy poprawne drzewo, każda z pomp o alfabecie zawartym w S może być pompowana w minimalnym drzewie o alfabecie S. A zatem drzewa = suma po minimalnych drzewach + pompowanie minimalnymi pompami o nie większych alfabetach. To daje natychmiast tw. Parikha.
c.n.d.

Uwaga: każdy semiliniowy zbiór da się uzyskać jako obraz Parikha języka bezkontekstwego, a nawet regularnego. Wystarczy np. dać aab(abccc)*(aaacc)* + acc(ac)*(abbbb)*, aby uzyskać {(1, 2, 0) + n(1, 1, 3) + k(3, 0, 2)} cup {(1, 0, 2) + n(1, 0, 1) + k(1, 4, 0)}.

Wykład 10

Przykład zastosowania:
1. Czy język {a^{2^n} | n in N} jest bezkontekstowy?
   NIE, bo jego obraz Parikha nie jest semiliniowy. A dlaczego nie jest?
2. Czy język {a^n b^{n^2} | n in N} jest bezkontekstowy?
   NIE, z tego samego powodu.

Ciekawostka
Zbiory semiliniowe są ciekawymi zbiorami. Po pierwsze są zamknięte na operacje boolowskie:

sumę, przecięcie, dopełnienie. Zamknięcie na sumę jest oczywiste, ale na przecięcie i dopełnienie dalece nie jest. Po drugie zbiory semiliniowe to te same, które są opisywalne w tzw. arytmetyce Presburgera, czyli logice pierwszego rzędu nad liczbami naturalnymi, gdzie możemy używać dodawania. Przykładowa formuła: phi(x, y, z) := (x+z = y) opisuje zbiór semiliniowy z przykładu 3. Ten wynik też nie jest oczywisty, chociaż oba powyższe wyniki dadzą się zrobić na kilku stronach.

Maszyny Turinga

Zaczynamy teraz trzeci dział na tym przedmiocie: maszyny Turinga.
Pytanie: co to jest mechaniczny proces? Maszyna Turinga to po prostu jeden z równoważnych modeli obliczeniowych, którymi jesteśmy w stanie zamodelować każde możliwe obliczenie. Tak się złożyło, że teraz na całym świecie używa się maszyn Turinga właśnie, ale jest wiele równoważnych modeli. Zdefiniował ją tak Alan Turing (1912-1954) w roku 1936.

Jest wiele różnych, niewiele się różniących wersji maszyn Turinga, my teraz zajmiemy się jednym konkretnym. Maszyna Turinga to taki bardziej skomplikowany automat dwukierunkowy. Tyle, że automat dwukierunkowy mógł chodzić tylko po słowie (w lewo, w prawie i nie ruszać się) i dodatkowo nic nie mógł pisać, a tylko czytał i w zależności od tego robił ruchy.
Natomiast maszyna Turinga będzie chodziła po taśmie, która jest nieskończona w obie strony. Na początku jest na niej napisane słowo wejściowe i maszyna zaczyna na lewym jego końcu. Dodatkowo maszyna może nie tylko czytać, ale też pisać.
Miejsca, które nie są zapisane literami słowa wejściowego w zapełnione są tzw. blankami, czyli specjalnymi literami oznaczającymi puste, nieodwiedzone miejsce. Maszyna ma też swój stan. W każdym ruchu maszyna patrzy na swój stan i literę napisanym na polu, na którym stoi. W zależności od tego zmienia stan, zapisuje na miejscu, na którym stoi jakąś literę (potencjalnie inną niż była) i idzie w którymś kierunku (lewo, prawo, stoi). Niektóre stany są akceptujące, a inne odrzucające. Jeśli maszyna wejdzie w taki stan, to akceptuje/odrzuca słowo wejściowe.

Formalnie, niedeterministyczna maszyna Turinga składa się z:

- alfabetu wejściowego Sigma
- zbioru stanów Q
- stanu początkowego q0
- zbioru stanów akceptujących F
- zbioru stanów odrzucających R
- alfabetu taśmowego Gamma, Sigma \subseteq Gamma
- symbolu B (blank) należącego do Gamma \setminus Sigma
- relacji przejścia: delta \subseteq Q x Gamma x Q x Gamma x {-1, 0, 1}

Konfiguracja maszyny jest postaci w q w', gdzie w to słowo przed głowicą, q to stan, a w' to słowo zaczynające się na komórce, na której jest głowica. Ignorujemy blanki (możemy założyć, że maszyna nigdy nie pisze blanków).
Konfiguracja początkowa maszyny to q0 w, czyli głowica maszyny stoi na pierwszej literze w, a stan to q0.

Tranzycja (q1, a1, q2, a2, k) oznacza: jeśli jesteś w stanie q1 i widzisz pod głowicą a1, to zmień stan na q2, zapisz pod głowicą a2 i przesuń głowicę o k komórek w prawo.

Bieg maszyny może się zakończyć (akceptując słowo bądź odrzucając), albo nie zakończyć. Zauważmy, że są różne sposoby jak może się nie zakończyć: może wpaść w cykl, ale może też np. cały czas w prawie, albo też inaczej. Ale to dla nas nie gra specjalnej roli. Może też być tak, że w pewnym staniu nie ma ruchu po pewnej literze, wtedy bieg jest odrzucany (ale staramy się robić maszyny tak, żeby tak się nie działo, wtedy po prostu przekierowujemy do stanu odrzucającego).

Słowo jest akceptowane jeśli istnieje chociaż jeden bieg akceptujący (w szczególności mogą być też inne biegi odrzucające). Język maszyny Turinga to zbiór słów akceptowanych. Maszyna jest deterministyczna jeśli relacja przejścia jest funkcją z (Q x Gamma) w (Q x Gamma x {-1, 0, 1}). Mówimy, że maszyna deterministyczna jest totalna jeśli zatrzymuje się (wchodzi do stanu z F cup R) dla każdego wejścia.

Taśma jest nieskończona, jednak w każdej chwili używana jest jedynie jej skończony kawałek. Więc nieskończonej taśmy nie należy traktować jako nieskończonej pamięci, a raczej jako dowolnie dużą pamięć, co już jest rozsądnym założeniem w rzeczywistym świecie (przy modelowaniu). Maszyny Turinga będą służyły za formalny model wszelkich możliwych obliczeń. Za jej pomocą można symulować każdy możliwy algorytm, ale tego nie widać od razu.

Przykłady

Spójrzmy na następujący przykład maszyny. Będzie ona rozpoznawać język palindromów nad {a, b}. Idea jest tak, że sprawdzi pierwszą literę, zapamięta ją w stanie i zamaże ją #, pójdzie na koniec i sprawdzi, czy ostatnia litera jest taka sama. Jeśli tak, to zamaże ją $ i pójdzie na początek, gdzie powtórzy to samo.

Sigma = {a, b}
Stany to: Q = {q0, qa, qb, ta, tb, p, ok, źle}
q0 - początkowy, ok - akceptujący, źle - odrzucający
Alfabet taśmowy Gamma = {a, b, B, #, $}, # będzie służyć do zamazywania początku, a $ końca (dałoby się to obsłużyć jednym symbolem, ale jest jaśniej co się dzieje).

Tranzycje:
(q0, a, qa, #, +1), (q0, b, qb, #, +1), (q0, B/$, ok, B/$, 0)
(qa, a/b, qa, a/b, +1), (qa, B/$, ta, B/$, -1)
(qb, a/b, qb, a/b, +1), (qb, B/$, tb, B/$, -1)
(ta, a, p, $, -1), (ta, b, źle, b, 0), (ta, #, ok, #, 0)
(tb, b, p, $, -1), (tb, a, źle, a, 0), (tb, #, ok, #, 0)
(p, a/b, p, a/b, -1), (p, #, q0, #, +1)

Teraz inny przykład, pokazujący, że maszyny Turinga mogą robić też bardziej skomplikowane rzeczy. Będzie ona rozpoznawać język {ww | w in Sigma*}.
Idea jest taka, że sprawdzi pierwszą literę, zapamięta ją w stanie i zamaże ją #, pójdzie w prawo i zgadnie w pewnym momencie, że tu jest początek drugiej połówki. Sprawdzi, czy litera się zgadza. Jeśli tak, to zamaże ją # i wróci po pierwszej połówki. Potem będzie robiła w kółko podobne rzeczy sprawdzając, czy litery w pierwszej połówki i z drugiej są takie same. Jak się zorientuje, że przeczytała wszystko i wszystko się zgadzało, to zaakceptuje, jak nie, to odrzuci.

Stany to: Q = {q1, q1a, q1b, p, q2, q2a, q2b, ra, rb, r, s, ok, źle}.
q1 - początkowy, ok - akceptujący, źle - odrzucający

Tranzycje:
(q1, a, q1a, #, +1), (q1, b, q1b, #, +1), (q1, B, ok, B, 0)
(q1a, a/b, q1a, a/b, +1), (q1a, a, p, #, -1), (q1a, B, źle, B, 0)
(q1b, a/b, q1b, a/b, +1), (q1b, b, p, #, -1), (q1b, B, źle, B, 0)
(p, a/b, p, a/b, -1), (p, #, q2, #, +1)
(q2, a, q2a, #, +1), (q2, b, q2b, #, +1), (q2, #, źle, #, 0)
(q2a, a/b, q2a, a/b, +1), (q2a, #, ra, #, +1)
(q2b, a/b, q2b, a/b, +1), (q2b, #, rb, #, +1)
(ra, #, ra, #, +1), (ra, a, r, #, -1), (ra, b/B, źle, b/B, 0)
(rb, #, rb, #, +1), (rb, b, r, #, -1), (rb, a/B, źle, a/B, 0)
(r, #, r, #, -1), (r, a/b, p, a/b, -1), (r, B, s, B, +1)
(s, #, s, #, +1), (s, a/b, źle, a/b, 0), (s, B, ok, B, 0)

Wydaje się, że ta maszyna faktycznie akceptuje język {ww | w in Sigma*}, ale może gdzieś być błąd, łatwo go zrobić w opisie maszyny. Tutaj zrobiliśmy tak precyzyjny opis to po, żeby go raz zobaczyć. Zazwyczaj będziemy po prostu mówić słowami co robi maszyna.

Maszyny wielotaśmowe
Maszyny Turinga, jak się okaże, opisują wszystko co trzeba, ale jednak są bardzo prostym (niskopoziomowym) modelem obliczeń. Jak widać wyżej nawet akceptacja prostych języków (jak {ww | w in Sigma*}) wymaga dość skomplikowanych maszyn. W związku z tym rozważa się nieraz ich uogólnienia, przy pomocy których nieco łatwiej opisać różne algorytmy. Rozważymy tutaj maszyny wielotaśmowe, które będą miały tę samą wyrażalność co maszyny Turinga (czyli będą akceptowały te same języki), ale będzie odrobinę łatwiej się nimi posługiwać.

Idea jest taka, że maszyna wielotaśmowa (nazywamy je wsadowymi) zamiast jednej taśmy ma wiele taśm. Na każdej taśmie jest jedna głowica, natomiast stan jest wspólny dla wszystkich taśm, tj. jeden dla całej maszyny. Jedna z tych taśm jest **wejściowa**, czyli można z niej tylko czytać, ale nic nie można tam zmieniać. Reszta taśm to tzw. taśmy **robocze**, na których można robić co się chce. Na początku na taśmie wejściowej jest słowo wejściowe, a wszystkie pozostałe są puste (głowice są na początku słowa i gdziekolwiek - bo to nie ma znaczenia, każde miejsce pustej taśmy jest takie samo).

Formalnie rzecz biorąc konfiguracja maszyny wsadowej o k+1 taśmach (wejściowej i k roboczych) jest postaci: Q x (Gamma* x Gamma*)^{k+1}, czyli stan oraz na każdej taśmie to, co po lewej i po (nieściśle) prawej od głowicy.
Natomiast relacja przejścia delta jest zawarta w Q x (Gamma^{k+1}) x Q x (Gamma^k) x {-1, 0, 1}^k.

Okazuje się, że maszyny wsadowe rozpoznają te same języki co maszyny jednotaśmowe.

Tw.
Maszyny jednotaśmowe i wsadowe rozpoznają te same języki.

Dowód
Łatwo zasymulować maszynę jednotaśmową na wsadowej dwutaśmowej. Po prostu najpierw kopiujemy taśmę wejściową na taśmę roboczą, a potem działamy tak jak jednotaśmowa na taśmie roboczej.
W drugą stronę, czyli symulacja maszyny wsadowej k+1 taśmowej na maszynie jednotaśmowej jest trochę trudniejsza, ale też nie bardzo. Powiedzmy, że maszyna wsadowa ma alfabet Gamma. Wtedy maszyna jednotaśmowa, która ją symuluje będzie miała alfabet (Gamma x {true, false})^{k+1} cup Sigma. Czyli jedna komórka taśmy odpowiada na k+1 komórek maszyny wsadowej, każda na innej taśmie. Wartość true/false odpowiada temu, czy na danej taśmie głowica jest właśnie na tej komórce. Musimy pamiętać pozycje głowic, bo teraz mamy tylko jedną. Sumujemy z Sigmą, żeby słowo wejściowe mogło być nad tym samym alfabetem. Teraz powiedzmy sobie co robi maszyna jednotaśmowa. Najpierw zamienia w = a1 a2 … ak na (a1-true, B-true, …, B-true) (a2-false, B-false, …, B-false) … (ak-false, B-false, …,B-false) i wraca na początek. Teraz symuluje zachowanie maszyny wielotaśmowej. Trzeba pokazać, że umie zasymulować jedną tranzycję, skupmy się na tym. Jak to zrobimy to już będzie jasne, że ciąg tranzycji też da się zasymulować. Żeby zasymulować tranzycję maszyny wsadowej trzeba wiedzieć jakie symbole znajdują się pod głowicami. Więc maszyna jednotaśmowa skanuje wejście z lewej do prawej i zapamiętuje w stanie te symbole. Jak przeczyta wszystkie to już wie która tranzycja powinna być zaaplikowana (albo która może, w przypadku maszyny niedeterministycznej). Wtedy wraca do początku, idzie znów od lewej do prawej i jeśli trafi na którejś taśmie na głowicę, to zmienia głowicę na tej taśmie zgodnie z tranzycją.
c.n.d.

Uwaga: zauważmy, że jeśli maszyna wsadowa była deterministyczna, to jednotaśmowa też, to nam się później przyda.

Maszyny deterministyczne
Pokażemy teraz, że maszyny deterministyczne mają tę samą siłę wyrazu co maszyny niedeterministyczne. To co prawda nie będzie oznaczało, że nie będziemy dalej rozważać maszyn niedeterministycznych, bo może się okazać, że do opisania tego samego języka

wystarczy mniejsza maszyna niedeterministyczna niż deterministyczna (tak jak w przypadku automatów).

Tw.
Każdy język rozpoznawany przez niedeterministyczną maszynę wsadową jest również rozpoznawany przez pewną deterministyczną maszynę wsadową.

Dowód
Na poziomie intuicyjnym symulacja polega na tym, że maszyna deterministyczna przeszukuje drzewo ścieżek maszyny niedeterministycznej. Konkretnie rzecz biorąc rozważmy jednotaśmową maszynę M niedeterministyczną o K tranzycjach. Skonstruujemy deterministyczną maszynę wsadową o 3 taśmach (2 roboczych) M' taką, że L(M') = L(M). Maszyna M' na pierwszej taśmie roboczej generuje coraz dłuższe ciągu liczb: n1, …, nm in {1, …, K}, które mają odpowiadać obliczeniom maszyny M. Następnie maszyna M' sprawdza na drugiej taśmie roboczej czy to obliczenie jest faktycznie akceptującym obliczeniem M. Jeśli tak, to akceptuje, a jeśli nie, to generuje następny ciąg liczb na pierwszej maszynie itd. Jeśli M ma pewien bieg akceptujący, to M' też i vice versa.

Wykład 11

Hierarchia Chomsky'ego
Noam Chomsky, znany lingwista, w 1956 opisał hierarchię typów gramatyk, znaną pod pojęciem hierarchii Chomsky'ego. Na każdym poziomie klasie gramatyk odpowiada klasa języków i klasa automatów.

Gramatyki typu 3, reguły: A → a, A → Ba
Języki regularne
Automaty skończone

Gramatyki typu 2, reguły: A → gamma
Języki bezkontekstowe
Automaty ze stosem

Gramatyki typu 1, reguły: alpha A beta → alpha gamma beta (gramatyki kontekstowe), gdzie alpha oraz beta to dowolne ciągi nieterminali i terminali
Języki kontekstowe
Maszyny Turinga z pamięcią ograniczoną liniowo

Gramatyki typu 0, reguły: alpha → beta (bez ograniczeń)
Języki **rekurencyjnie przeliczalne** (albo języki **częściowo rozstrzygalne**)
Maszyny Turinga (niedeterministyczne, koniecznie kończące się)

Ta hierarchia moim zdaniem nie jest szczególnie ważna, bardziej z powodów historycznych warto ją znać. Warto też rozumieć jak różne klasy mają się do siebie.

Języki i problemy rozstrzygalne

Mówimy, że maszyny Turinga (jedno/wielotaśmowe, niedeterministyczne/deterministyczne) mają być modelem programów. Chcemy jednak żeby każdy program się kończył no i działał w zdeterminowany sposób. Dlatego sensownym modelem dla programów są tak naprawdę deterministyczne maszyny Turinga, które dla każdego wejścia kończą bieg. Inny słowy mówimy, że są totalne, albo posiadają własność stopu.

Języki, które są rozpoznawane przez takie maszyny to języki **rekurencyjne** lub też języki **rozstrzygalne**. Nie jest prawdą, że każdy język rekurencyjnie przeliczalny (częściowo rozstrzygalny) jest też rekurencyjny (przeliczalny). Idea jest taka, że maszyna może działać w nieskończoność i może nie być maszyny, która jest jej równoważna, ale zawsze się kończy. W tej chwili tego nie pokażemy, bo w ogóle nie wiemy, czy jest jakiś język nierozstrzygalny.

Języki a problemy

Język to podzbiór Sigma*. Problem decyzyjny to taki, dla którego dla każdego wejścia mamy odpowiedzieć TAK lub NIE. Przykładowe problemy decyzyjne to:

- dany graf G, wierzchołki s, t, czy s → t?
- dana liczba n, czy jest pierwsza?
- dany automat A, czy L(A) jest pusty?

Zauważmy, że problem decyzyjny można też widzieć jako język, mianowicie jako zbiór tych wejść, dla których odpowiedź jest TAK. Czyli powiedzmy problem osiągalności w grafie to zbiór słów postaci opis(G)#s$t takich, że z wierzchołka s da się osiągnąć wierzchołek t w grafie G. Od tej pory będziemy utożsamiali problemy decyzyjne z językami, warto się do tego przyzwyczaić.

Inne równoważne modele

Jest wiele modeli równoważnych maszynom Turinga:

- deterministyczne maszyny Turinga
- niedeterministyczne/deterministyczne maszyny wsadowej
- gramatyki typu 0 (bez dowodu)
- automaty wielostosowe (pewnie zobaczymy to jeszcze)
- automaty z kolejką
- automaty z wieloma licznikami
- obwody logiczne
- itp. itd.

Rozstrzygalność a częściowa rozstrzygalność

Jak mówiliśmy nie jest tak, że każdy problem częściowo rozstrzygalny jest rozstrzygalny. To samo w sobie jest bardzo ciekawym faktem: nie każdy problem da się rozwiązać! Pokażemy to poniżej.

Natomiast prawdą jest następujący fakt (nie jest bardzo ważny).

Fakt

Każdy problem L taki, że zarówno L jak i Sigma* \ L są częściowo rozstrzygalne jest rozstrzygalny.

Dowód
Niech M1 rozpoznaje L, a M2 rozpoznaje Sigma* \ L, obie deterministyczne (bo możemy tak założyć, jak pokazaliśmy wcześniej). Robimy M, która rozpoznaje L i się kończy. M symuluje naraz M1 i M2 na wejściu. Jak M1 zaakceptuje, to akceptuje, a jak M2 zaakceptuje, to odrzuca. Dla każdego wejścia w maszyna M się zatrzyma, bo albo w in L (wtedy M1 się zatrzyma) albo w not in L (wtedy M2 się zatrzyma).

Kodowanie i maszyna uniwersalna
Łatwo sobie wyobrazić, że każdy obiekt, z którym mamy do czynienia można zakodować jako ciąg liter. Np. graf o n wierzchołkach można zakodować jako macierz sąsiedztwa. A automat można zakodować jako kodowania Q, Sigma, F, q0, delta po kolei. Podobnie można również zakodować maszynę Turinga w słowie.

Bardzo przydatnym pojęciem jest **maszyna uniwersalna**. Bierze się ona właśnie z obserwacji, że dla każdej maszyny Turinga M istnieje pewne słowo, które ją koduje (Sigma, Gamma, Q, F, …), nazwijmy je kod(M). A zatem możemy rozważyć język L = {(kod(M), w) | M akceptuje w}. Łatwo zauważyć, że język L można rozpoznać maszyną Turinga M'. Ta maszyna po prostu symuluje M na słowie w. Taka maszyna M' nazywa się maszyną uniwersalną. Czyli każdy problem rozstrzygalny da się sprowadzić do uniwersalnego problemu, który to jest językiem maszyny uniwersalnej.

Nierozstrzygalność problemu stopu
Dość zaskakującym odkryciem jest to, że istnieją problemy, dla których nie ma żadnego algorytmu, który by je rozwiązywał, teraz to pokażemy. Takich problemów jest wbrew pozorom całkiem sporo. Najprostszym chyba dowodem na to jest następujący argument: problemów jest tyle ile podzbiorów Sigma*, czyli continuum. Natomiast algorytmów (czy też maszyn Turinga) jest Alef0, czyli mniej. Muszą więc istnieć problemy decyzyjne, dla których nie ma algorytmów. Nie jest ta jednak specjalnie ciekawe, bo chcemy raczej wiedzieć, że istnieją problemy, które da się w skończony sposób opisać i nie ma dla nich algorytmów (mówimy, że są nierozstrzygalne). I to jak się okaże jest dość powszechne. Najbardziej chyba kanonicznym (i znanym) problemem nierozstrzygalnym jest tzw. **problem stopu**:

Dane: deterministyczna maszyna Turinga M, słowo w
Pytanie: czy M zatrzymuje się na w

Gdyby problem stopu był rozstrzygalny, to rozstrzygalny byłby też **problem należenia**:

Dane: deterministyczny maszyna Turinga M, słowo w
Pytanie: czy w in L(M)

Zrobilibyśmy bowiem maszynę M', która działa tak jak M na początku. Jak M akceptuje, to M' też, a jak M odrzuca, to M' się pętli. A więc M akceptuje w ⇔ M' zatrzymuje się na w. Pokażemy więc, że problem należenia jest nierozstrzygalny.

Najpierw przypomnijmy znaną zagadkę o fryzjerze w miasteczku X, który strzyże wszystkich, którzy nie strzygą się sami. Takiego fryzjera nazwiemy superfryzjerem. Pytanie brzmi: czy superfryzjer strzyże sam siebie - szybko dochodzimy do wniosku, że żadna opcja nie jest możliwa. Czyli superfryzjer nie może istnieć.

Dowód
Niech miasteczko X zamieszkane będzie przez kody maszyn Turinga.
Powiemy, że kod(M1) strzyże kod(M2) jeśli M1 akceptuje kod(M2).
Przypuśćmy, że problem należenia jest rozstrzygalny, skonstruujemy superfryzjera w miasteczku X. Jeśli problem należenia jest rozstrzygalny, to znaczy, że istnieje maszyna N taka, że
L(N) = {(kod(M), w) | M akceptuje w}. Rozważmy teraz nową maszynę F taką, że jeśli wejście nie jest kodem żadnej maszyny Turinga, to odrzuca, a jeśli wejście do kod(M), to akceptuje jeśli M nie akceptuje kod(M). Mając N możemy łatwo skonstruować F, po sprawdzeniu, czy wejście jest kodem symuluje N na parze (kod(M), kod(M)), a potem neguje odpowiedź. Czyli F akceptuje wejścia tych maszyn Turinga (strzyże je), które nie akceptują samych siebie (czyli nie strzygą się). Czyli F jest superfryzjerem w miasteczku X. Sprzeczność, bo superfryzjer nie może istnieć.
c.n.d.

Nierozstrzygalność ciąg dalszy
Przyjrzyjmy się jeszcze raz temu co zrobiliśmy powyżej. Pokazaliśmy tak naprawdę, że problem należenia jest nierozstrzygalny. Pokazaliśmy poza tym, że problem należenia da się przeformułować jako problem stopu. A więc gdyby problem stopu był rozstrzygalny, to problem należenia też. Zatem problem stopu musi być również nierozstrzygalny.

Tę taktykę będziemy stosować cały czas do pokazywania, że pewne inne problemy są nierozstrzygalne. Będziemy pokazywać, że gdyby one były rozstrzygalne, to problem stopu (albo inny nierozstrzygalny) też byłby rozstrzygalny.
Formalnie rzecz biorąc ujmiemy to w pojęcie **redukcji**.

Niech K to język zawarty w Sigma*, a L język zawarty w Gamma*.
Redukcja z K do L to jest funkcja **obliczalna** f: Sigma* → Gamma* taka, że
w in K ⇔ f(w) in L
Zauważmy, że jeśli L jest rozstrzygalny i istnieje redukcja z K do L, to K też jest rozstrzygalny. Istotnie, robimy następująco. Jak dostajemy w in Sigma*, to aplikujemy do niego f. Potem patrzymy na f(w) i pytamy, czy należy do L. To jest rozstrzygalne. Ale wiemy, że f(w) in L ⇔ w in K, czyli odpowiedzieliśmy na pytanie, czy w in K. Czyli K też rozstrzygalny.

A więc redukcja **z** problemu nierozstrzygalnego do naszego (który rozważamy) pokazuje, że nasz jest nierozstrzygalny.

Pokażemy teraz, że różne problemy są nierozstrzygalne:

1. pustość języka maszyny: dana maszyna M, czy L(M) jest pusty
2. pustość przecięcia języków dwóch gramatyk: dane gramatyki bezkontekstowe G1, G2, czy L(G1) \cap L(G2) jest pusty
3. uniwersalność gramatyki: dana gramatyka bezkontekstowa G, czy L(G) = Sigma*
4. problem odpowiedniości Posta (PCP - Post Correspondence Problem): dane pary słów (u1, v1), …, (un, vn), czy istnieje taki ciąg indeksów i1, …, ik, że ui1 ui2 … uik = vi1 vi2 … vik

No to dowodzimy po kolei nierozstrzygalności.

1. Robimy redukcję z problemu należenia do problemu pustości języka. Dana maszyna M i słowo w, skonstruujemy maszynę M' taką, że w in L(M) ⇔ L(M') niepusty. M' będzie zależała od M i od w. Maszyna M' po prostu ignoruje wejściowe słowo, potem wpisuje słowo w na taśmę i symuluje M. Jeśli w in L(M), to L(M') = Sigma*, wpp. L(M') jest pusty. Czyli to jest redukcja do problemu niepustości, gdybyśmy umieli rozstrzygać pustość, to należenie też byśmy rozstrzygnęli. Podobnie można pokazać zresztą, że problem pełności maszyny Turinga jest nierozstrzygalny (można by potem zanegować wejście).

Wykład 12

Kontynuujemy pokazywanie nierozstrzygalności różnych problemów.

2. Robimy redukcję z problemu pustości języka maszyny M. Dla M skonstruujemy dwie gramatyki G1, G2 takie, że L(M) = pusty ⇔ L(G1) \cap L(G2) pusty. Zrobimy G1 i G2 takie, że L(G1) \cap L(G2) będzie równy dokładnie poprawnym kodowaniom biegów maszyny M. Teraz powiemy co to jest kodowanie biegu. To jest #c1#rev(c2)#...#ck# (co druga konfiguracja odwrócona), gdzie wszystkie ci mają równą długość. Alfabetem to Gamma \cup Gamma x Q \cup #, litera z Gamma oznacza komórkę taśmy, a litera z Gamma x Q oznacza komórkę taśmy nad którą jest głowica (wpisujemy tam stan). Zobaczmy, że łatwo zrobić gramatykę, która rozpoznaje w#rev(w). Można też zrobić gramatykę, która rozpoznaje ci#rev(c(i+1)), gdzie c(i+1) powstaje z ci przez jeden krok maszyny. Podobnie z rev(ci)#c(i+1). Niech więc K to język pierwszego typu (ci#rev(c(i+1))), a L drugiego typu (rev(ci)#c(i+1)). Robimy G1 i G2 tak, żeby
   L(G1) = # (Sigma\#)* # (L#)* \cup (#K)*#
   L(G2) = (#K)* # (Sigma\#)* # \cup # (Sigma\#)* # (L#)* (Sigma\#)* #
   Pierwsze części odpowiadają za kodowania nieparzystej długości, a drugie parzystej. Nietrudno zobaczyć, że faktycznie L(G1) \cap L(G2) rozpoznaje to, co trzeba.

3. Robimy redukcję z problemu należenia do problemu pustości języka. Skonstruujemy gramatykę G taką, że L(M) jest pusty ⇔ L(G) = Sigma*. Konkretnie rzecz biorąc L(G) będzie zawierać wszystkie słowa oprócz poprawnych kodowań akceptujących biegów M. Zauważmy najpierw, że język dop(w#w) jest bezkontekstowy. Możemy podobnie

pokazać, że język dop(w#next(w)) jest bezkontekstowy. A więc robimy teraz G takie, że

L(G) = (Sigma* #)* dop(w#next(w)) (# Sigma*)* \cup ((Sigma\#)* \ w0 B*)

Łatwo zobaczyć, że rzeczywiście L(G) to wszystko oprócz poprawnych kodowań akceptujących biegów M.

4. Teraz PCP. Najpierw przykład, 3 pary: (b, bbb), (babbb, ba), (ba, a).
   Rozwiązanie to (2, 1, 1, 3), bo
   babbb b b ba = ba bbb bbb a
   Będziemy robili redukcję z problemu należenia do PCP. Najpierw jednak pokażemy, że jeśli umiemy rozwiązywać PCP, to też coś więcej, tzn. PCP z ustaleniem która para jest pierwsza w rozwiązaniu. Rozważmy PCP (u1, v1), …, (un, vn). Robimy teraz nową instancję PCP (na przykładzie): (*b*, *b*b*b), (b*, *b*b*b), (b*a*b*b*b*, *b*a), (b*a*, *a), ($, *$). Łatwo zauważyć, że nowe PCP ma rozwiązanie wtw stare miało rozwiązanie zaczynające się od pary (b, bbb). Czyli PCP z ograniczeniem jest nie trudniejsze niż ogólne PCP. Teraz zredukujemy problem należenia do PCP z ograniczeniem. Zrobimy takie PCP, by rozwiązania reprezentowały tylko poprawne kody akceptujących obliczeń maszyny M na słowie w, tzn. $c0#c1#...#ck$. Najpierw przeróbmy naszą maszynę tak, że na samym końcu, jak już jest w stanie akceptującym, to zamazuje wszystko specjalnym symbolem S i przechodzi na początek do specjalnego stanu f, nigdy też w trakcie biegu ten S nie jest użyty (będzie nam łatwiej zrobić to PCP).
   Idea:
   ($, $c0#), (a, a), (B, B), (#, #), (abc, a'b'c') jeśli a'b'c' powstaje w jednym kroku maszyny z abc, (#f, eps), (S, eps), (S$, $).
   Nietrudno zauważyć, że rzeczywiście rozwiązania naszego PCP odpowiadają biegom maszyny.

Złożoność obliczeniowa

Maszyny Turinga przydają się nie tylko do uściślenia faktu, że istnieje jakiś problem
Dobrze nadają się też do sformalizowania pojęcia "szybki program", dla różnych wersji szybkości. Zajmiemy się teraz wstępem do złożoności obliczeniowej, która zadaje sobie pytanie jak bardzo złożone są problemy, to znaczy jak "szybki" jest najszybszy program, który może je rozwiązać. Oczywiście jak zwykle dbamy tu o zgrubne oszacowanie szybkości, asymptotyczne itd.

Rozważmy jakąś funkcję f: N → N taką, że f(n) \geq log(n). Wtedy oznaczamy:
DTIME(f) = problemy, które da się rozwiązać w **czasie** f(n) maszyną **deterministyczną**
NTIME(f) = problemy, które da się rozwiązać w **czasie** f(n) maszyną **niedeterministyczną**
DSPACE(f) = problemy, które da się rozwiązać w **pamięci** f(n) maszyną **deterministyczną**
NSPACE(f) = problemy, które da się rozwiązać w **pamięci** f(n) maszyną **niedeterministyczną**

Czas działania maszyny to liczba kroków, a **pamięć** to liczba komórek, które zostały odwiedzone. Jasne jest, że czas jest zawsze nie mniejszy (asymptotycznie) niż pamięć, bo jeśli

zrobiliśmy K kroków to odwiedziliśmy maksymalnie K+|w| komórek pamięci (słowo wejściowe tu dodaje stały czynnik).

Definiujemy szereg bardzo ważnych klas przy użyciu powyższych metod:
P (= PTIME) = suma po k naturalnych DTIME(n^k)
NP = suma po k naturalnych NTIME(n^k)
PSPACE = suma po k naturalnych DSPACE(n^k)
NSPACE = suma po k naturalnych NSPACE(n^k)
EXPTIME = suma po k naturalnych DTIME(2^{n^k})
NEXPTIME = suma po k naturalnych NTIME(2^{n^k})
L = DSPACE(log(n))
NL = NSPACE(log(n))

Zachodzi L ⊆ NL ⊆ P ⊆ NP ⊆ PSPACE = NPSPACE ⊆ EXPTIME ⊆ NEXPTIME ⊆ …
Co ciekawe dzisiejsza nauka bardzo niewiele wie na temat ścisłego zawierania.
Wiadomo jedynie, że:

- L mniejsze od PSPACE
- P mniejsze od EXPTIME
- ogólnie, że dla f mniejszej istotnie od g zachodzi DTIME(f) mniejsze od DTIME(g) oraz DPSACE(f) mniejsze od DSPACE(g). To przy założeniu, że f i g rozsądne. Jeśli g większa od f^2 to już chyba starczy. Więcej na przedmiocie teoria złożoności.

P vs NP
Problem, czy P jest równe NP jest uznawany za obecnie najważniejszy problem informatyki teoretycznej. W 2000 roku Instytut Matematyczny Claya ogłosił 7 problemów uznanych przez nich za obecnie najważniejsze problemy matematyki. Za rozwiązanie każdego z nich fundowana jest nagroda miliona dolarów. A to oczywiście jedynie część chwały. Jest tam 6 problemów ściśle matematycznych i jeden z informatyki teoretycznej, właśnie, czy P jest równe NP. Zresztą moim zdaniem różnica między informatyką teoretyczną a matematyką jest umowna.

Przypomnijmy więc, NP to są te problemy, które są rozpoznawalne w czasie wielomianowym na niedeterministycznej maszynie Turinga. NP od Nondeterministic Polynomial, a na pewno nie od Non Polynomial, jak niektórzy mogą myśleć. Tutaj ważna jest intuicja tego niedeterminizmu, a chodzi o zgadywanie. Żeby to dokładnie zrozumieć przyjrzyjmy się najbardziej znanemu problemowi z klasy NP (o którym nie wiadomo, czy należy do P).

Jest to problem spełnialności. Dana jest formuła logiki zdaniowa używająca n zmiennych: x1, …, xn. Ta formuła może używać negacji, koniunkcji i alternatywy (wszystko pozostałe da się wyrazić).
Problem spełnialności (SAT od satisfiability) dla danej formuły zdaniowej phi pyta, czy istnieje takie wartościowanie zmiennych, że phi jest spełniona.

Jak pokazać, że problem SAT jest w klasie NP? Trzeba zaprojektować maszynę niedeterministyczną, która go rozwiązuje. Pomysł jest taki. Maszyna M na początku ma dwa przejścia, odpowiadające opcjom: x1 = 0 oraz x1 = 1. Potem ma dwa odpowiadające x2 = 0 oraz x2 = 1. Itd. aż na końcu ma dwa odpowiadające opcjom xn = 0 lub xn = 1. Powiedzmy, że maszyna zapisuje to wartościowanie na taśmie. Następnie maszyna sprawdza w czasie wielomianowym, czy to wartościowanie, które jest odpowiednie dla danego biegu jest poprawne. Jeśli jest to akceptuje, wpp. odrzuca. Zobaczmy, że jeśli istnieje jakieś wartościowanie spełniające, to istnieje też bieg mu odpowiadający i ten bieg będzie akceptujący. Wpp. nie ma dobrego biegu.

Jaki jest sens tej maszyny? Często się mówi, że zgaduje ona wartościowanie i potem sprawdza, czy dobrze zgadła. Tak się zazwyczaj myśli o problemach w NP: jest to zgadnięcie pewnego wielomianowego obiektu (początek biegu, który jest niedeterministyczny), a potem weryfikacja w czasie wielomianowym tego, czy to zgadnięcie było dobre.

Rozważmy inny znany problem w klasie NP. Jest to problem Komiwojażera. Dany jest graf skierowierowany (pełny) z wagami (liczbami naturalnymi) na krawędziach oraz pewna liczba T. Wagi określają czas przejazdu między wierzchołkami (miastami). Pytanie, czy komiwojażer (przedstawiciel firmy podróżujący w celu zdobywania klientów) może objechać wszystkie miasta w czasie co najwyżej T. Zakładamy przy tym, że nigdy nie odwiedza 2 razy tego samego miasta. Łatwo pokazać, że ten problem jest w NP. Po prostu maszyna zgaduje odpowiednią trasę komiwojażera, a potem weryfikuje, że jest dobra.

NP-zupełność

Okazuje się, że istnieją problemy “najtrudniejsze” w klasie NP. Takie problemy będziemy nazywać NP-zupełnymi. No bo każdy problem w P jest też w NP, warto zajmować się tymi problemami w NP, które są naprawdę trudne. Powiemy, że problem jest NP-zupełny jeśli jest w NP oraz jest NP-trudny. Natomiast problem L jest NP-trudny jeśli:

każdy problem w NP da się zredukować redukcją w czasie wielomianowym do L.

To jest bardzo podobne do tego, co robiliśmy przy nierozstrzygalności. Mianowicie wyobraźmy sobie, że jest problem L, który jest NP-zupełny. Gdybyśmy pokazali, że L jest w P, to już łatwo pokazać, że NP = P. Weźmy bowiem dowolny problem K z klasy NP. Istnieje redukcja w czasie wielomianowym K do L. Przypomnijmy, że redukcje wielomianowa K do L to funkcja (teraz obliczalna w czasie wielomianowym) f taka, że

w in K ⇔ f(w) in L

A więc żeby stwierdzić, czy w należy do K liczymy f(w) i patrzymy, czy należy do L. Ale na to pytanie da się znaleźć odpowiedź w czasie wielomianowym, bo założyliśmy, że L należy do P.

A zatem jeśli jakiś problem NP-zupełny jest w P, to NP = P. I to jest właśnie zaleta problemów NP-zupełnych. Apriori nie wiadomo wcale, czy jakiś problem NP-zupełny istnieje. Przecież tak jak dla każdej liczby naturalnej jest większa liczba naturalna to dla każdego problemu w NP mógłby istnieć od niego trudniejszy, ale wciąż w NP. Okazuje się, że jednak istnieją problemy

NP-zupełne. Co więcej jest ich całkiem sporo (jak już będziemy mieli jeden, to łatwo wskazać więcej sprowadzając ten jeden do nich, jak w nierozstrzygalności).

Jest to słynne twierdzenie Cooka z 1971 roku.

Tw. Cooka
Problem SAT jest NP-zupełny.

Dowód
Weźmy pewien problem L należący do NP.
Pokażemy jak dla słowa wejściowego w skonstruować taką instancję SATa phi_(w, L), że

w in L ⇔ phi_(w, L) jest spełnialna

Ponieważ L należy do NP, to istnieje pewna niedeterministyczna maszyna M_L, która rozpoznaje język L. A więc w należy do L wtw. gdy istnieje bieg akceptujący maszyny M_L na słowie w. Formuła phi_(w, L) będzie właśnie kodować to, że istnieje taki bieg.

Niech słowo w ma długość n. Najpierw określmy jakie zmienne występowały w formule:

- x(i, j, a): po i krokach maszyny na j-tej komórce taśmy jest litera a
- x(i, q): po i krokach maszyna jest w stanie q
- x(i, j): po i krokach głowica maszyny jest na j-tej komórce taśmy

Zauważmy, że skoro i jest ograniczone przez O(n^k), to j też, a więc liczba zmiennych jest ograniczona przez O(n^2k). Można to ukonkretnić, np. 17 n^9, więc wiadomo ile tych zmiennych napisać. Teraz chcemy napisać formułę, która będzie spełnialna wtw. gdy istnieje poprawny bieg M_L na w:

- niesprzeczność (w każdej chwili dokładnie 1 stan, w każdej chwili głowica na dokładnie jednym miejscu, w każdej chwili w każdej komórce dokładnie jedna litera)
- konfiguracja początkowa (na odpowiednich komórkach odpowiednie litery, głowica na początku, stan początkowy)
- konfiguracja końcowa (istnieje i oraz q w F taki, że x(i, q))
- poprawność przejść (jeśli nie ma gdzieś głowicy to się nic nie zmienia, jeśli jest, to istnieje tranzycja (czyli skończona alternatywa po tranzycjach) taka, że okolica się tak właśnie zmienia)

Widać, że dla słowa w skonstruowaliśmy w czasie wielomianowym formułę, która jest spełnialna wtw. gdy w należy do L. Czyli to jest faktycznie redukcja w czasie wielomianowym co kończy dowód.
c.n.d.

Pokażemy teraz, że kilka innych problemów jest NP-zupełnych. Tak naprawdę tych problemów jest nieskończenie wiele, a bardzo wiele naturalnych.

Najbardziej chyba znany to jest 3-SAT, czyli pewne wzmocnienie SATa.
Powiemy, że formuła zdaniowa jest w postaci 3-CNF (conjunctive normal form) jeśli jest koniunkcją **klauzul**:

phi = C1 ∧ C2 ∧ … ∧ Cn,

gdzie każda klauzula jest alternatywą co najwyżej 3 **literałów**:

Ci = x3 v neg(x5) v x6,

gdzie literały to zmienne lub ich negacje.
Przykład to:

(x1 v neg(x2) v x3) ∧ (neg(x3) v x1 v x4) ∧ (x5 v neg(x6) v x2) ∧ (x6 v neg(x2) v neg(x1))

Ta formuła jest spełnialna, gdyż wartościowanie: x1 = 1, x2 = 1, x6 = 1, a pozostałe dowolne spełnia formułę.

Można tak zmodyfikować, dowód Tw. Cooka, żeby pokazać, że konstruujemy formułę w postaci CNF. Stąd mamy twierdzenie:

Tw
SAT dla formuł w postaci CNF jest NP-trudny.

Fakt
3-SAT jest NP-zupełny

Dowód
Łatwo pokazać, że 3-SAT jest w NP, po prostu zgadujemy wartościowanie spełniające i sprawdzamy, czy działa. Teraz żeby pokazać, że 3-SAT jest NP-trudny wystarczy zrobić redukcję z dowolnego problemu NP-trudnego. Wtedy dla dowolnego K w NP będziemy mieli redukcję: K → inny NP-trudny → nasz, czyli będziemy mieli redukcję do naszego. Zrobimy redukcję z SATa dla formuł w postaci CNF, więc z niego zrobimy redukcję.

Wystarczy pokazać, że formułę zdaniową phi w postaci CNF można przekształcić w czasie wielomianowym do formuły zdaniowej psi takiej, że:

- phi spełnialna ⇔ psi spełnialna
- psi jest w postaci 3-CNF

Robię następująco: jeśli mam klauzulę (l1 v l2 v … v lm) to robię nową zmienną x i przerabiam ją na koniunkcję dwóch klauzul:
(l1 v l2 v … v l(m-2) v x) ∧ (neg(x) v x(m-1) v xm).
Łatwo zobaczyć, że ta koniunkcja jest spełnialna wtw. gdy wejściowa klauzula jest spełnialna.
Mogę tak robić produkując nowe klauzule wielkości 3 i zmniejszając moją do czasu gdy m <= 3.
A więc sprowadzę w ten sposób do wielomianowo większej postaci 3-CNF.

c.n.d.

Wykład 13

Na początek musimy jeszcze udowodnić tw. Cooka, bo nie zrobiliśmy tego poprzednio.

Pokażmy jeszcze jedną przykładową redukcję. W problemie kliki mamy dany graf G i liczbę n (daną unarnie) i pytamy, czy w grafie G jest klika Kn jako podgraf indukowany.
Pokażemy następujący fakt.

Fakt
Problem kliki jest NP-zupełny.

Dowód
Łatwo pokazać bycie w NP: zgadujemy n wierzchołków i sprawdzamy, że to jest istotnie klika.
Teraz pokażemy NP-trudność, poprzez redukcję z 3-SATA.
Weźmy formułę: phi = C1 ∧ C2 ∧ … ∧ Ck o zmiennych x1, …, xn.
Stworzymy graf G i liczbę k taką, że phi spełnialna wtw. G ma klikę wielkości k.
Dla każdego literału w klauzuli tworzymy oddzielny wierzchołek. Mamy ich więc co najwyżej 3k, czyli wielomianowo wiele. Łączymy dwa literały jeśli nie są w tej samej klauzuli oraz jeśli nie są tą samą zmienną, tylko raz zanegowaną, a raz nie. Twierdzimy, że w tym grafie jest klika wielkości k ⇔ phi jest spełnialna. Pokażemy to. Jeśli phi jest spełnialna, to istnieje wartościowanie takie, że w każdej klauzuli jest jakiś literał, który jest spełniony. Te literały z pewnością są połączone krawędziami, czyli tworzą klikę wielkości k. Z drugiej strony jeśli istnieje klika wielkości k, to możemy stworzyć wartościowanie, które przypisze tym zmiennym takie wartości, że są spełnione (nie ma tam x oraz neg(x)), czyli phi jest spełnialna.
c.n.d.

Znane problem NP-zupełne

Można w podobny sposób pokazać, że jeszcze wiele problemów jest NP-zupełnych.
Znane przykłady to:

- czy w danym grafie istnieje cykl Hamiltona? (problem cyklu Hamiltona)
- problem komiwojażera
- dane n przedmiotów o wartościach wi i objętościach oi oraz plecak o całkowitej objętości O. Jak zapakować plecak, by miał maksymalną wartość? (problem plecakowy)
- dany graf, czy da się pokolorować jego wierzchołki na 3 kolory tak, by sąsiednie wierzchołki miały zawsze różne kolory? (3 kolorowanie)
- dany zbiór liczb całkowitych, czy istnieje jego niepusty podzbiór o sumie zero? (problem sumy podzbioru)
- itp. itd.

Równe, czy nie?

Jest bardzo wiele takich problemów i ludzie bardzo wiele myśleli nad tym, czy P = NP.
Większość naukowców uważa, że P jest różne od NP, ale są też tacy, którzy przychylają się bardziej do hipotezy, że P = NP, a my, ludzkość, po prostu bardzo mało wiemy o algorytmach.

Ja osobiście też uważam, że P jest różne od NP, a nie umiemy tego pokazać m.in. dlatego, że w ogóle prawie nic nie umiemy udowodnić jeśli chodzi o tę dziedzinę. Prawie że w ogólnie nie są znane techniki, które pokazują, że jakieś problemy nie są rozwiązywalne w czasie wielomianowym. W ogóle znanych jest bardzo mało dolnych ograniczeń w złożoności obliczeniowej (czyli dziedzinie, która zajmuje się klasyfikacją problemów względem ich skomplikowania, m.in. właśnie przyporządkowywania do różnych klas złożoności). Od lat 70-tych niewiele zrozumieliśmy. Jest pewien postęp, ale głównie w tym, że wiemy, że niektóre techniki nie pozwolą nam na pokazanie, że P jest różne od NP.

Próby obejścia

Są różne praktyczne podejścia do sprawy. Skoro jest wokół nas tyle problemów NP-trudnych to należy jakoś się do nich zabrać. Wybiły się następujące podejścia:

- algorytmy randomizowane (z dużym prawdopodobieństwem i szybko rozwiązujemy problem). Jednak jest niewiele problemów, które dają się zrobić w ten sposób, a nie znamy algorytmu w P.
- algorytmy aproksymacyjne (rozwiązujemy problem mniej więcej, np. znajdujemy nie najkrótszą ścieżkę komiwojażera, ale taką, która jest nie dłuższa niż najkrótsza + 1% trasy). W praktyce takie algorytmy wystarczą i okazuje się, że często dadzą się zrobić szybko.
- algorytmy wykładnicze o niskich podstawach. Np. zamiast 10^n robimy (1,3)^n. To pozwala na trochę większe n, ale też nie tak znowu duże. Według mojej (niewielkiej) wiedzy na ten temat to jest głównie teoretycznie rozważane.
- algorytmy heurystyczne. Działają zazwyczaj w miarę nieźle. Dość znane są tzw. SAT-solvery, czyli programy, które dla większości danych instancji SATa rozwiązują go szybko. To są po prostu heurystyki, które mówią - znajdź zmienną, która występuje w wielu miejscach i najpierw ją zgadnij, rozważaj klauzule z mniejszą liczbą literałów itd. Przy sprytnych metodach większość instancji SATa da się szybko zrobić.

Kryptologia

Jednak okazuje się, że nie wszystko da się szybko rozwiązać nawet heurystycznie. Kryptologia opiera się na tym, że pewne rzeczy da się szybko zrobić w jedną stronę, a w drugą nie wiemy jak to zrobić szybko. Jednym z najbardziej popularnych takich problemów jest mnożenie dwóch liczb. Jak mamy dane dwie duże (powiedzmy 200-cyfrowe) liczby pierwsze, to możemy łatwo, w czasie O(200^2) pomnożyć je. Jednak jak mamy ten iloczyn i chcemy go rozdzielić na czynniki pierwsze (dwa), to nie są znane żadne, nawet heurystyczne, metody, żeby to zrobić szybko. Nawet w sposób randomizowany, przybliżony (cokolwiek by to miało znaczyć w tym przypadku) itd. Czyli mówimy, że nie umiemy robić szybko faktoryzacji (tj. rozkładu na czynniki pierwsze). Dygresja: algorytmy na komputerach kwantowych zaczęły się od algorytmu Shora, który pokazał, że na (jeszcze nie zbudowanym w rozsądnej skali) komputerze kwantowym da się robić szybko faktoryzację (w czasie wielomianowym, przy użyciu losowości). Na tym się opiera kryptologia właśnie. Inny podobny problem to problem logarytmu dyskretnego. Jak mamy a, k, n, to umiemy obliczyć szybko b = a^k mod n. Ale jak mamy a, b, n,

to nie umiemy szybko obliczyć k takiego, że b = a^k mod n. Ten problem też jest często używany w kryptologii.

Zauważmy, że oba problemy są łatwo w NP. Dla faktoryzacji zgadujemy rozkład i łatwo sprawdzamy, że jest ok, a dla logarytmu dyskretnego zgadujemy k i też łatwo sprawdzamy. Nie wiadomo o tym żeby były NP-zupełne (zazwyczaj to się łatwo pokazuje, że duża szansa, że nie są).

Czyli gdyby P = NP, to byśmy mieli problem z kryptologią, bezpieczeństwem itd. Podobnie zresztą gdyby nauczono się budować komputery kwantowe, wtedy faktoryzacja i logarytm dyskretny byłyby rozwiązywalne łatwo. Powstała już dziedzina, która się nazywa kryptologia post-kwantowa i zajmuje się projektowaniem bezpiecznych szyfrów i protokołów na wypadek zbudowania komputera kwantowego.

PSPACE
Inną dość często rozważaną klasą jest PSPACE, czyli zbiór problemów, które dadzą się rozwiązać na maszynie Turinga w wielomianowej pamięci.
Ciekawe jest twierdzenie Savitcha, które mówi, że w tym wypadku niedeterminizm wcale nie dodaje siły wyrazu.

Tw. (Savitch)
PSPACE = NPSPACE

Pominiemy jego dowód.

W klasie PSPACE są również problemy zupełne, tj. PSPACE-zupełne. To są problemy, które są w PSPACE oraz są PSPACE-trudne, czyli każdy problem z PSPACE da się do nich sprowadzić za pomocą redukcji wielomianowej.

Typowy problem PSPACE-zupełny to QBF (Quantified Boolean Formulas), dane jest formuła postaci:

exists_x1 forall_x2 exists_x3 … forall_x(n-1) exists_xn phi(x1, …, xn),

gdzie phi używa tylko zmiennych x1, …, xn. Pytanie, czy jest ona prawdziwa. Zobaczmy, że to jest trochę podobne do SATa, który pyta, czy formuła postaci:

exists_x1 exists_x2 exists_x3 … exists_x(n-1) exists_xn phi(x1, …, xn)

jest prawdziwa.

Inny problem PSPACE-zupełny to uniwersalność automatu niedeterministycznego. Czyli: dany automat niedeterministyczny A, czy L(A) = Sigma*?
Wiele problemów PSPACE-zupełnych ma związek z grami. Na przykład ustalanie gracza posiadającego strategię wygrywającą w uogólnionych warcabach jest PSPACE-zupełne. Intuicja jest taka, że jeden gracz odpowiada za exists, a drugi za forall. Tzn. gracz pierwszy wygrywa

wtw. gdy istnieje jego ruch taki, że dla każdego ruchu przeciwnika istnieje jego ruch t., że dla każdego ruchu przeciwnika itd.

NL

Dość znana jest też klasa NL - problemy rozwiązywalne niedeterministycznymi maszynami w pamięci logarytmicznej. Tutaj rozważamy redukcje w pamięci logarytmicznej, a nie w czasie wielomianowym, bo te w czasie wielomianowym nie miałyby za bardzo sensu. Wtedy bowiem już sama redukcja mogłaby rozwiązać problem.

Najbardziej znanym problemem NL-zupełnym jest osiągalność w grafie: dany graf skierowany G i dwa jego wierzchołki s, t, pytanie, czy w grafie G z wierzchołka s da się osiągnąć wierzchołek t (tj. czy istnieje ścieżka).
Fajny jest algorytm osiągalności, który działa w NL. Po prostu zgadujemy ścieżkę, pamiętamy gdzie jesteśmy i idziemy krok dalej. Pamięć logarytmiczna nam wystarczy, bo pamiętamy tylko numer wierzchołka. Jednak musimy jakoś skończyć. Obserwacja jest taka, że jeśli da się osiągnąć t z s, to da się osiągnąć jakąś ścieżką o długości co najwyżej n. Czyli możemy kończyć przeszukiwanie po n krokach. Czyli robimy tak: zgadujemy kroki i liczymy je. Jak dojdziemy do n, to przerywamy i nie akceptujemy. Jeśli jest ścieżka z s do t, to ją znajdziemy, wpp. oczywiście nie. Używa pamięć jest logarytmiczna.

Używamy tu istotnie niedeterminizmu, bo zgadujemy ścieżkę. Innym dużym problemem otwartym jest, czy L = NL. Wydaje się, że raczej nie, ale naukowcy i tu nie są pewni.

Inne klasy złożoności

Wiadomo jedynie o niektórych klasach, że są różne. Tzn. nie wiadomo, czy P jest różne od PSPACE, ani czy L jest różne od P, ale wiadomo, że:

- L jest różne od PSPACE
- P jest różne od EXPTIME

W ogólności są twierdzenia o hierarchii (czasowej i pamięciowej), które mówią z grubsza, że dla rozsądnych funkcji f jeśli mamy funkcję g > f^2, to wtedy:

- DTIME(f) ściśle zawarte w DTIME(g)
- DSPACE(f) ściśle zawarte w DSPACE(g).

To niewiele, ale zawsze coś.

A więc hierarchia jest nieskończona. Tzn. PTIME istotnie zawarte w EXPTIME, a to w 2-EXPTIME, a to w 3-EXPTIME itd. Są też problemy nierozwiązywalne w żadnym czasie n-EXPTIME, ale rozwiązywalne i nawet jeszcze trudniejsze. Ale to już temat na zupełnie inną opowieść. Trochę o tym na Złożoności Obliczeniowej.

Podsumowanie

Na tym przedmiocie uczyliśmy się o modelach obliczeń. Największy nacisk położyliśmy na trzy: automaty skończone, automaty ze stosem (i gramatyki) oraz maszyny Turinga. Jest też wiele innych, choć te trzy są rzeczywiście chyba najbardziej popularne.

Znając to, co wiemy możemy iść dalej, w kierunku precyzyjnego klasyfikowania problemów na trudniejsze i łatwiejsze. To zahacza trochę o filozofię i ogólnie zrozumienie świata. Dla zainteresowanych polecam np. te teksty:

- http://www.deltami.edu.pl/temat/matematyka/teoria_liczb/2016/12/27/Prawda_o_matematykach/
- http://www.deltami.edu.pl/temat/informatyka/algorytmy/2013/12/30/Na_granicy_mozliwosci/

Proszę o wypełnienie ankiet po tym przedmiocie. Szczególnie cenne będą dla mnie uwagi tekstowe. Dziękuję za cały semestr, to dla mnie było ciekawe doświadczenie, mam nadzieję, że dla Państwa również.